企業級 AI 系統開發技術叢書

Harness Engineering

大型語言模型的運行環境工程

從代理迴圈、上下文工程、工具介面
到多代理、評估、安全與成本的系統性著作

Agent = Model + Harness

基於截至 2026 年 8 月之公開工程文獻編纂

進階技術讀本·面向具 IT 背景之讀者

本書為技術教學讀本, 內容以截至 2026 年 8 月之公開第一手工程文獻 (Anthropic、OpenAI、Google、Microsoft 之工程部落格與規格文件, 及可復現之基準與研究) 為依據編纂, 關鍵論斷均於各章末列出出處。書中所涉產品之介面、功能與版本演化迅速, 讀者引用具體產品細節時, 應以最新官方文件為準; 本書力求準確, 但不對因產品變動所致之時效性差異負責。

書中程式碼示例以說明為目的, 採最小實現, 省略生產環境所需之完整錯誤處理與安全加固。

版本: 第一版·2026 年 8 月

目錄

前言

二〇二四年底,Anthropic 在其工程部落格發表〈Building Effective Agents〉一文, 系統性地區分了工作流 (workflow) 與代理 (agent), 並提出「增強型大型語言模型」(augmented LLM) 作為代理系統的基本構件。當時,「harness」這個詞尚未成為行業術語——人們談論的是 scaffolding(鷹架)、agent framework(代理框架), 或者籠統的 agentic system。

十四個月之後, 情況徹底改變。二〇二六年二月五日,Mitchell Hashimoto(HashiCorp 共同創辦人、Ghostty 作者) 發表〈My AI Adoption Journey〉, 將自己使用 AI 開發的最高階段命名為「Engineer the Harness」; 六日之後,OpenAI 發表題為〈Harness Engineering: Leveraging Codex in an Agent-First World〉的官方文章, 描述七名工程師如何在五個月內以接近零人手撰寫的方式產出約一百萬行程式碼。同年三月,Anthropic 發

表〈Harness Design for Long-Running Application Development〉; 四月, 其 Managed Agents 架構文章寫下了整個領域最重要的一句話:

> 「Harness 編碼了一些假設, 而這些假設會隨着模型進步而過時。」(Harnesses encode assumptions that go stale as models improve.)

八月,Microsoft 將 Agent Framework 中的生產運行時層正式命名為「Harness」並使其正式上市 (GA)。至此,「Harness Engineering」從一個社群俚語, 演變為一個有明確工程內涵、有相互競爭的設計學派、有可量化效果的正式學科。

本書是一部關於這門學科的系統性著作。它試圖回答三個問題:

第一,**Harness 是什麼**。一個大型語言模型本身只能生成文字; 使它成為能夠讀寫檔案、執行指令、操作瀏覽器、完成長時程任務的代理的, 是圍繞模型建造的整套運行環境——工具介面、上下文管理機制、記憶體系、執行沙箱、驗證迴圈、權限體系、狀態持久化與可觀測性基礎設施。

這套環境就是 harness。本書第一部與第二部將逐一拆解這些組件的設計原理。

第二,**Harness 為什麼重要**。證據是量化的: 同一個模型,換一套 harness,SWE-bench Verified 的得分可以相差二十個百分點以上;Anthropic 曾以純粹的 harness 改進,使同一個 Claude 3.5 Sonnet 模型的得分從 33% 提升至 49%; 在 Terminal-Bench 的公開排行榜上, 同一模型在不同 harness 之下的分數跨度可達三十個百分點。模型能力決定了上限,harness 決定了你實際拿到多少。本書第四部將詳細討論評估、可觀測性、安全與成本這四個維度上 harness 的決定性作用。

第三,**Harness 應該怎樣設計**。這個領域在二〇二六年出現了兩個相互對立又相互補充的學派:OpenAI 所代表的「累積式」(accretive) 學派主張, 每當代理犯錯, 就以規則、Lint、CI 檢查將該錯誤永久性地工程化消除,harness 隨時間不斷增厚;Anthropic 所代表的「削減式」(subtractive) 學派則主張,harness 中的每個組件都是對模型當前缺陷的補償, 應當隨模型進步而被壓力測試並刪除。理解這兩種學派的張力, 是做出正確架構決策的前提。本書第三部與第五部將展開這場辯論, 並提供可操作的決策框架。

本書的讀者

本書假定讀者具備軟件工程背景: 你應當熟悉 HTTP、JSON、版本控制、容器化部署與基本的分散式系統概念, 並且寫過或至少讀得懂 Python 與 TypeScript。本書不會解釋什麼是 API, 不會用日常比喻來稀釋技術內容, 也不會迴避諸如 KV cache、旋轉位置編碼 (RoPE)、OAuth 2.1 授權框架、seccomp-bpf 或 constrained decoding 這類術語——它們會在首次出現時被準確定義, 然後被當作已知概念使用。

本書刻意保持廠商中立。Anthropic、OpenAI、Google、Microsoft 的第一手工程文獻都會被大量引用, 因為這個領域的最佳文獻正是這些機構的工程部落格與規格文件; 但本書的分析框架不綁定任何一家的產品。所有產品細節均標注了資料來源與時間點——這個領域演化極快, 讀者應當意識到, 具體產品的介面與功能會過時, 而設計原理的半衰期要長得多。

本書的結構

全書分為五部, 共二十三章, 另附三個附錄。

第一部「基礎」(第一至四章) 建立理論地基:harness engineering 的定義與歷史、Transformer 推理的微觀機制、上下文窗口的退化物理學 (context rot), 以及工作流與代理的架構分類學。

第二部「Harness 的組件」(第五至十一章) 逐一拆解 harness 的七大組件: 工具介面 (agent-computer interface)、指令與記憶層級、檢索與 compaction、執行沙箱、Model Context Protocol、Agent Skills, 以及驗證迴圈與 hooks。

第三部「架構模式」(第十二至十六章) 處理系統層面的組合問題: 單代理工作流模式、多代理系統的收益與代價、跨上下文窗口的長時程代理設計、狀態持久化與框架選型, 以及對 Claude Code、Codex CLI 等產品級 harness 的架構解剖。

第四部「品質、安全與營運」(第十七至二十章) 覆蓋生產化的四大支柱: 評估工程 (包括 pass^k 與 harness 敏感

性)、可觀測性 (OpenTelemetry GenAI 語義約定)、安全工程 (prompt injection、lethal trifecta、CaMeL 與縱深防禦), 以及成本工程與 token 經濟學。

第五部「實踐與未來」(第二十一至二十三章) 包含一個從零開始、約六百行 Python 的完整 harness 實作, 組織層面的採用策略, 以及對 harness 自動化與消亡論的前瞻分析。

附錄提供術語表、帶注釋的文獻清單, 以及可直接用於架構評審的 harness 設計檢查清單。

致謝與說明

本書內容以截至二〇二六年八月的公開文獻為準, 所有關鍵論斷均可追溯至章末所列的第一手資料。引文若原文為英文, 均由作者譯出並附原文於必要處。書中程式碼示例以 Python 3.11+ 與 TypeScript 5+ 撰寫, 均為說明用途的最小實現, 省略了生產環境所需的完整錯誤處理。

願這本書幫助你在模型與應用之間, 建造那一層真正決定成敗的工程。

Part I

第一部基礎

第一章從模型到代理:Harness Engineering 的誕生

1.1 一個定義

先給出本書的核心定義, 再逐步論證它。

代理 (agent)= 模型 (model)+ Harness。

模型是一個以自回歸方式逐 token 生成文字的函數: 給定一段輸入序列, 它輸出下一個 token 的機率分佈。無論這個模型的參數量是八十億還是數兆, 無論它經過多少輪強化學習後訓練 (post-training), 它的介面本質上只有一個: 文字進, 文字出。模型不能讀取你的檔案系統, 不能執行 shell 指令, 不能呼叫 REST API, 不能記住上一次對話, 甚至不能知道現在幾點。

Harness 是圍繞這個文字函數建造的全部運行環境, 它至

少包含以下七類組件:

1. **代理迴圈 (agent loop)**——驅動模型反覆推理的控制流: 將模型輸出解析為行動, 執行行動, 將結果回饋給模型, 直到任務完成或觸發終止條件。
2. **工具介面 (tool interface / agent-computer interface, ACI)**——模型可調用的可執行能力集合, 連同其 schema、描述、輸入驗證與輸出格式化。
3. **上下文管理 (context management)**——決定每一次推理時上下文窗口中放什麼、不放什麼: 系統提示、指令檔、檢索機制、compaction、記憶體系。
4. **執行環境 (execution environment)**——工具實際運行的場所: 檔案系統、shell、沙箱、網絡出口控制、憑證管理。
5. **驗證機制 (verification)**——判斷代理的行動是否正確的回饋來源: 測試、lint、截圖比對、規則檢查、評審模型。
6. **權限與安全 (permissions & safety)**——決定哪些行動可以自動執行、哪些需要人類批准、哪些被絕對禁止的策略層。
7. **狀態與可觀測性 (state & observability)**——跨會

> 話的持久化、檢查點、恢復機制, 以及追蹤、日誌、指標。

Microsoft 工程師 Wes Steyn 在 2026 年 Agent Framework Harness 正式上市時的表述是對這個定義最簡潔的概括:「模型本身只能生成文字」——是 harness 把它變成代理。

Harness Engineering, 就是對上述七類組件進行系統性設計、度量與迭代的工程學科。它的核心信條由 Mitchell Hashimoto 在 2026 年 2 月給出:

> 「每當你發現代理犯了一個錯, 你就投入時間去工程化一個解決方案, 使代理永遠不再犯這個錯。」

而它最重要的反面警語, 則來自 Anthropic 同年四月的 Managed Agents 架構文章:

> 「Harness 編碼了一些假設, 而這些假設會隨着模型進步而過時。」

一門學科同時擁有一條建設性信條和一條解構性警語, 說明它已經成熟到出現內部辯證——這正是本書要展開的核

心張力。

1.2 為什麼是「文字函數」不夠: 一個具體的推演

考慮一個看似簡單的任務:「修復這個 repository 中 issue #4128 描述的 bug。」

一個裸模型 (bare model) 面對這個任務, 能做的只有: 根據它在訓練資料中見過的類似程式碼, 猜測 bug 可能是什麼樣子, 然後生成一段看起來合理的修補。它看不到這個 repository 的實際程式碼, 不知道 issue #4128 的內容, 無法運行測試來確認修補是否有效, 更無法提交 commit。它的輸出是一段「統計上似然的文字」, 而不是一個「被驗證過的工程行動」。

現在逐步為它添加 harness 組件, 觀察能力如何湧現:

- 加入**檔案讀取工具**與 **grep 工具**: 模型可以實際定位到出錯的函數, 而不是猜測。
- 加入 **shell 工具**: 模型可以運行測試套件, 觀察失敗的斷言與 stack trace。

- 加入**檔案編輯工具** (以精確字串替換而非整檔重寫的形式): 模型可以做出最小化修改, 降低誤傷面。
- 加入**代理迴圈**: 模型可以「修改 → 運行測試 → 觀察失敗 → 再修改」地迭代, 直到測試轉綠。
- 加入**上下文管理**: 當測試輸出長達數萬 token 時,harness 截斷或摘要它, 防止淹沒模型的注意力。
- 加入**權限系統**:`rm -rf`、`git push --force` 這類指令被攔截, 等待人類批准。
- 加入**驗證關卡**:harness 在代理宣稱完成時強制運行完整 CI, 不通過則拒絕結束會話。

每加一層, 系統的可靠性都發生質變。而值得注意的是: **模型自始至終沒有變**。這就是 harness engineering 的第一性原理: 在模型能力固定的前提下, 系統能力仍有巨大的、可工程化的提升空間。

這個提升空間有多大? 第十七章將詳細展開證據, 這裏先給三個數字:

- 2024 年 10 月,Anthropic 在〈Raising the bar on SWE-bench Verified〉中報告, 對同一個 Claude 3.5 Sonnet 模型, 僅透過重新設計 harness(一個只

有持久化 bash 與字串替換編輯器兩個工具的極簡 scaffold, 加上對工具描述的密集打磨), 得分從 33% 提升到 49%——十六個百分點, 零模型改動。

- 2025 年 6 月,Epoch AI 的分析指出,「一個好的 scaffold 可以將表現提升多達 20%」; 同一個 DeepSeek R1-0528 模型, 在 Inspect scaffold 下得 33%, 在 Agentless scaffold 下得 57.6%。
- 2026 年,Terminal-Bench 的公開排行榜上有一百四十多個「harness+ 模型」組合條目, 同一個前沿模型在不同 harness 下的得分跨度可以超過三十個百分點。

1.3 術語考古: 從 Scaffolding 到 Harness Engineering

「Harness」一詞的演化史本身就是這門學科的形成史, 值得精確記錄。

第一階段 (2023–2024): 評估社群的「scaffolding」。最早需要系統性地「把模型接上真實環境」的, 是做危險能力評估與自主性評估的機構。METR(前 ARC Evals) 的 task

standard 使用「task harness」與「scaffolding」指稱驅動模型完成評估任務的程式框架。這個時期的主流詞是 scaffolding——鷹架, 暗示它是暫時性的、輔助性的結構。

第二階段 (2024 年 12 月):Anthropic 的「Agent-Computer Interface」。〈Building Effective Agents〉沒有使用 harness 一詞, 而是創造了 ACI(agent-computer interface) 這個對照 HCI(human-computer interface) 的術語, 強調「為模型設計工具介面應當投入與為人類設計介面同等的心力」。該文附錄二記載了一個標誌性細節: 他們在 SWE-bench 代理上花在優化工具上的時間, 超過了花在整體提示上的時間; 其中一個改動——強制要求絕對路徑——消滅了一整類相對路徑錯誤。這種「以介面設計消滅錯誤類別」的思路, 即日本製造業所稱的 poka-yoke(防呆設計), 後來成為 harness engineering 的方法論核心。

第三階段 (2025 年): 廠商語彙的常態化。 OpenAI 的 Codex CLI 在其文檔中自稱「coding harness」;Anthropic 將 Claude Agent SDK 描述為「a powerful, general-purpose agent harness」;2025 年 11 月 26 日,Anthropic 發表〈Effective Harnesses for Long-

Running Agents〉, 首次將 harness 放進文章標題。同年,「context engineering」(上下文工程) 一詞也由 Anthropic 的〈Effective Context Engineering for AI Agents〉(2025 年 9 月 29 日) 正式化, 學術界隨後以一篇覆蓋一千四百餘篇論文的綜述 (arXiv:2507.13334) 確立了它的研究地位。

第四階段 (2026 年 2 月):「Harness Engineering」的結晶。這個短語在兩週之內完成定名。2 月 5 日,Mitchell Hashimoto 發表〈My AI Adoption Journey〉, 把個人 AI 採納歷程分為六個階段, 其中第五階段命名為「Engineer the Harness」, 並給出了那條著名的信條; 他描述自己的 AGENTS.md 檔案「每一行都對應一個曾經出現過的糟糕代理行為, 而它幾乎完全解決了所有這些行為」。2 月 11 日,OpenAI 發表〈Harness Engineering: Leveraging Codex in an Agent-First World〉, 以完整案例展示這門學科的威力: 一個從三人擴至七人的團隊, 五個月內產出約一百萬行程式碼、約一千五百個 pull request, 平均每位工程師每天合併 3.5 個 PR, 估計零行程式碼是人手撰寫的。

第五階段 (2026 年 3 月至今): 主流化與產品化。 An-

thropic 的〈Harness Design for Long-Running Application Development〉(3 月 24 日) 通篇使用 harness design; 其 Managed Agents 服務 (4 月 8 日) 把 harness 作為一個獨立版本化、可拋棄的系統組件來架構;Microsoft 在 Build 2026 後將 Agent Framework 的生產運行時直接命名為「Harness」, 並於 8 月正式上市。至此,harness 從隱喻變成了產品名詞。

這段歷史揭示了一個重要事實:harness engineering 不是某一家公司的營銷概念, 而是多個獨立團隊在相同的工程壓力下收斂出的相同結論。當 Anthropic、OpenAI、Microsoft、Google 以及開源社群 (opencode、Cline、goose 等) 在 2025 至 2026 年間各自獨立地實現出幾乎同構的組件集——指令檔、子代理、hooks、沙箱、compaction、MCP 整合、可觀測性——收斂本身就是這些設計正確性的最強證據。

1.4 兩大學派: 累積式與削減式

2026 年的 harness engineering 存在兩種對立的設計哲學, 理解它們是本書後續所有架構討論的前提。

累積式 (Accretive) 學派

以 OpenAI 的〈Harness Engineering〉一文為代表。其邏輯是: 代理的每一次失敗都暴露了環境的一個缺陷, 正確的回應是**永久性地消除該缺陷**——寫一條 AGENTS.md 規則、加一個自訂 linter、建一道 CI 關卡、造一個更好的驗證工具。harness 因此隨時間單調增厚, 成為組織的核心資產。OpenAI 團隊的具體實踐包括:

- 一份約一百行的 AGENTS.md, 作為「地圖而非千頁說明書」, 指向一個結構化的 `docs/` 目錄樹;
- 以自訂 linter 與 CI **機械性強制**的分層架構 (Types→Config→Repo→Service→Runtime→UI), 使代理不可能寫出違反分層的程式碼;
- 「黃金原則」(golden principles) 作為週期性後台清理任務——一種架構層面的垃圾回收;
- 「代理可讀性」(agent legibility) 信條:「任何代理在運行時無法在上下文中取得的東西, 實際上等於不存在」;
- 給代理開放可觀測性存取 (Chrome DevTools Protocol、LogQL、PromQL、TraceQL), 使其能自行

診斷生產問題。

其組織哲學被濃縮為兩句話:「人類掌舵, 代理執行」(Humans steer. Agents execute.) 與「修正是廉價的, 等待是昂貴的」。

削減式 (Subtractive) 學派

以 Anthropic 的 2026 年系列文章為代表。其邏輯是:harness 中的每個組件都是對「模型當前做不到某事」這一假設的編碼, 而模型在快速進步, **因此每個組件都應被定期壓力測試, 一旦假設失效就刪除該組件**。最有力的實證來自 Anthropic 自己的迭代:

2025 年 11 月的〈Effective Harnesses for Long-Running Agents〉為了對抗 Claude Opus 4.5 的「上下文焦慮」(context anxiety, 模型在感知到上下文將盡時傾向草草收尾), 設計了每會話只做一個 feature、頻繁重置上下文、以 JSON 檔案鎖定 feature 清單等機制。四個月後,〈Harness Design for Long-Running Application Development〉(2026 年 3 月) 報告:Opus 4.6 大幅緩解了上下文焦慮, 於是他們**刪除了** sprint 合約機制, 把逐

feature 驗收改回單次整體評估。文章的結論句應當被每個架構師背誦:

> 「Harness 中的每個組件都編碼了一個關於模型自身做不到什麼的假設, 而這些假設值得被壓力測試。」

兩派的調和

這兩種哲學並不真正互斥, 它們適用於 harness 的不同層。本書的立場, 將在第二十二、二十三章詳細論證, 可以預先概括為:

- **針對環境的組件應當累積**: 更快的測試、更準的 linter、更完整的文檔樹、更好的可觀測性——這些是對「你的系統」的投資, 不依賴模型的缺陷, 永不過時。
- **針對模型缺陷的組件應當可刪除**: 上下文重置策略、防止模型刪除測試的檔案格式選擇、強制分步驟的 sprint 機制——這些是對「模型當前弱點」的補丁, 應當帶着「預期壽命」被設計, 並在每次模型升級時重新評估。

區分一個組件屬於哪一類, 是 harness 架構評審中最有價值的一個問題。

1.5 Harness Engineering 與相鄰學科的邊界

與 prompt engineering 的關係。 Prompt engineering 關注單次推理的輸入文字設計;context engineering 將其推廣為「在每一次推理時刻, 策展上下文窗口中 token 集合」的持續過程;harness engineering 再進一步, 涵蓋上下文之外的一切——執行環境、驗證、權限、狀態、可觀測性。三者是嚴格的包含關係:prompt ⊂ context ⊂ harness。

與 MLOps 的關係。 MLOps 管理模型的訓練、部署與監控;harness engineering 假定模型是外部提供的黑盒 (通常經 API 取用), 管理的是模型的**使用環境**。兩者在可觀測性與評估上有工具重疊, 但關注的對象不同:MLOps 度量模型,harness engineering 度量「模型 × 環境」的複合系統。第十七章將引用 Anthropic〈Demystifying Evals for AI Agents〉(2026 年 1 月) 的定義: 代理評估的單位是

「harness 與模型作為一個整體」。

與傳統軟件工程的關係。這是最深刻的一層關係。本書將反覆展示:harness engineering 的幾乎每一個有效技術, 都是傳統軟件工程實踐在非確定性組件上的重新應用——版本控制作為代理的記憶與撤銷機制、CI 作為驗證迴圈、least privilege 作為權限模型、檢查點與冪等性作為容錯設計、程式碼評審作為 LLM-as-judge 的原型。Anthropic 在設計長時程 harness 時的明確方法論就是「複製高效軟件工程師每天在做的事」。Harness engineering 不是對軟件工程的取代, 而是軟件工程對一種新型不可靠組件——機率性的語言模型——的擴展。

1.6 本書的方法論立場

在進入技術細節之前, 申明三個貫穿全書的立場:

第一, 證據優先於敘事。這個領域充斥着演示視頻與軼事。本書只採信三類證據: 第一手工程文獻 (廠商工程部落格中帶量化數據的文章)、規格文件 (MCP、AGENTS.md、Agent Skills 等) 與同行評審或可復現的研究 (SWE-bench、Terminal-Bench、METR、

Chroma context rot 報告等)。所有數字均標注來源與時間。

第二, 原理優先於產品。產品介面的半衰期以月計——本書寫作期間,MCP 規格經歷了五個修訂版,OpenAI 的 Assistants API 走完了從 deprecated 到關閉的全程。本書組織材料的方式是先原理後實例: 先講清楚「為什麼需要 compaction」的注意力經濟學, 再看各家產品如何實現它。

第三, 承認不確定性。削減式學派的警語同樣適用於本書自身: 書中的某些工程建議, 本質上是對 2026 年模型能力邊界的補償, 它們會過時。凡屬此類, 本書都會明確標注; 凡屬更持久的原理 (注意力預算的物理限制、驗證不對稱性、least privilege), 也會明確論證其持久性的來源。

1.7 本章小結

- 代理 = 模型 + harness。模型是文字函數;harness 是使其成為代理的全部運行環境, 包含代理迴圈、工具介面、上下文管理、執行環境、驗證、權限、狀態與可觀測性七類組件。

- Harness 對系統表現的影響是量化可證的: 同模型換 harness, 基準得分可移動十六至三十個百分點。
- 「Harness engineering」一詞在 2026 年 2 月由 Hashimoto 與 OpenAI 先後定名, 但其實踐脈絡可追溯至 METR 的評估 scaffolding 與 Anthropic 2024 年的 ACI 概念。
- 兩大學派: 累積式 (OpenAI, 錯誤應被永久工程化消除) 與削減式 (Anthropic,harness 組件編碼的模型假設會過時, 應定期刪減)。針對環境的投資應累積, 針對模型缺陷的補償應可刪除。
- Harness engineering 是傳統軟件工程在非確定性組件上的擴展, 而非其替代。

本章參考文獻

1. Anthropic, “Building Effective Agents”, 2024-12-19. https://www.anthropic.com/engineering/building-effective-agents
2. Anthropic, “Raising the bar on SWE-bench Verified”, 2024-10-30. https://www.anthropic.com/news/swe-bench-sonnet

3. Mitchell Hashimoto, “My AI Adoption Journey”, 2026-02-05. https://mitchellh.com/writing/my-ai-adoption-journey
4. OpenAI, “Harness Engineering: Leveraging Codex in an Agent-First World”, 2026-02-11. https://openai.com/index/harness-engineering/
5. Anthropic, “Effective Harnesses for Long-Running Agents”, 2025-11-26. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents
6. Anthropic, “Harness Design for Long-Running Application Development”, 2026-03-24. https://www.anthropic.com/engineering/harness-design-long-running-apps
7. Anthropic, “Scaling Managed Agents: Decoupling the Brain from the Hands”, 2026-04-08. https://www.anthropic.com/engineering/managed-agents
8. Epoch AI, “What skills does SWE-bench Veri-

fied evaluate?”, 2025-06-13. https://epoch.ai/publications/what-skills-does-swe-bench-verified-evaluate

9. InfoQ, “Microsoft Agent Framework Harness Reaches General Availability”, 2026-08. https://www.infoq.com/news/2026/08/agent-framework-harness-ga/ # 第二章推理的微觀機制:Token、注意力與生成迴圈

Harness 的一切設計決策, 最終都受制於模型推理的微觀機制。不理解 tokenization 就無法正確估算成本; 不理解 KV cache 就無法解釋為什麼 prompt caching 要求前綴穩定; 不理解自回歸解碼就無法理解為什麼工具呼叫的 JSON 會格式錯誤, 以及 constrained decoding 為什麼能根治它。本章以 harness 工程師需要的精度——不多也不少——重建這套機制。

2.1 Tokenization: 模型世界的原子

大型語言模型不直接處理字符, 而是處理 token——由子詞分詞器 (subword tokenizer, 主流實現為 BPE,byte-pair

encoding) 將文字切分而成的整數序列。現代前沿模型的詞表 (vocabulary) 規模在十萬至二十餘萬之間。

Harness 工程師需要記住的經驗值:

- 英文約 0.75 個單詞對應一個 token; 一個 token 平均約四個字符。
- 中文 (無論繁簡) 通常一至兩個字符對應一個 token, 密度顯著低於英文——同樣語義的內容, 中文消耗的 token 往往比英文多 40% 至 70%。這對以中文撰寫系統提示或指令檔的團隊是實際的成本與注意力預算問題。
- 程式碼的 token 效率介乎兩者之間, 但高度依賴語言與格式; 縮排空白、長識別符、重複樣板都在消耗 token。
- JSON 的轉義 (引號、換行的 `\"` 與 `\n`) 使其比等價的純文字或 Markdown 更昂貴, 也更容易在生成時出錯——這是第五章討論工具輸出格式選擇時的一個核心論據。

Tokenization 還解釋了一些反直覺的模型失敗: 字符級操作 (倒轉字串、數字母) 困難, 是因為模型「看到」的是

token 而非字符; 罕見識別符被切成多個碎片後, 模型對其的表徵品質下降。Harness 設計中「使用語義化識別符而非 UUID」的建議 (見第五章), 部分根源即在此。

2.2 Transformer 注意力: 上下文的計算結構

推理時,decoder-only Transformer 對輸入序列的處理分兩個階段。

Prefill 階段: 整個輸入 (上下文窗口內的全部 token) 被一次性並行處理。每一層的自注意力 (self-attention) 為每個 token 計算 query、key、value 向量; 每個 token 的表徵透過對所有先前 token 的 key 做縮放點積、softmax 歸一化後加權求和 value 而更新。這一階段的計算量與序列長度呈平方關係——n 個 token 之間存在 n^2 對注意力關係。

Decode 階段: 模型逐 token 生成。每生成一個新 token, 只需為它計算注意力, 但要對上下文中所有既有 token 的 key/value 做點積——因此那些 key/value 被緩存起來, 即

KV cache。

KV cache 是 harness 成本工程的核心槓桿 (第二十章詳述), 這裏先建立兩個結論:

1. **Prompt caching 的本質是跨請求復用 KV cache**。供應商 (Anthropic、OpenAI、Google 均已提供) 允許把輸入的穩定前綴的 KV cache 保留一段時間, 後續請求命中時只需為新增部分做 prefill, 成本可降至常規輸入價格的十分之一左右。其工程含義是: **上下文必須設計為「穩定前綴 + 可變尾部」的結構**。系統提示、工具定義、指令檔放在最前且保持字節級穩定; 任何對前綴的改動 (哪怕一個字符) 都會使其後全部緩存失效。這直接約束了 harness 的上下文佈局: 這也是為什麼各家 harness 都把「對話歷史的增量追加」作為預設, 而對「修改歷史中段」(如 compaction) 的觸發保持謹慎。
2. **長上下文的成本是超線性的**。隨着上下文增長, 每個新生成 token 的注意力計算與記憶體頻寬開銷線性增長, 而整個任務的總成本 (所有輪次的累計輸入) 呈平方級增長。一個在 200k token 上下文中運行的代理迴圈, 每一輪都在為全部歷史付費。這是

compaction、子代理隔離、工具結果卸載等技術 (第七章) 的經濟學基礎。

位置編碼與外推極限

Transformer 本身對序列順序無感, 位置資訊由位置編碼注入。當代主流是旋轉位置編碼 (RoPE, rotary position embedding) 及其變體; 要把在較短序列上訓練的模型擴展到更長上下文, 需對 RoPE 做內插或頻率縮放 (NTK-aware scaling、YaRN 等)。這類外推是有損的: 模型在其訓練分佈以外的位置上, 位置分辨率下降。Anthropic 在〈Effective Context Engineering〉中對 context rot 的機理概括正是:n^2 注意力關係被攤薄、訓練資料偏向短序列導致「處理全上下文依賴的專用參數更少」、位置編碼內插使模型對位置的理解劣化。這三點構成第三章上下文退化現象的機制解釋。

2.3 自回歸解碼與採樣: 非確定性的來源

模型輸出的每一步是一個在整個詞表上的機率分佈, 解碼策略決定如何從中選取:

- **Greedy / temperature=0**: 總選最大機率 token。即便如此, 由於浮點運算的非結合性與批次處理的動態性, 雲端 API 仍不保證嚴格可重現。
- **Temperature 採樣**: 溫度 T 縮放 logits 後採樣,T 越高分佈越平。
- **top-p(nucleus)/ top-k**: 截斷低機率尾部後再採樣。

Harness 工程師必須內化的事實是: **代理系統的非確定性是乘法性的**。單次呼叫的輸出方差, 經過代理迴圈的多輪放大——第 t 輪的輕微措辭差異改變第 t+1 輪的上下文, 進而改變行動選擇——使得同一任務的兩次運行可能走出完全不同的軌跡。這正是第十七章 pass^k 指標 (k 次全部成功的機率, 對照 pass@k 的至少一次成功) 存在的理由, 也是驗證機制 (第十一章) 必須獨立於軌跡、只驗收終態的理由。

Constrained decoding: 把格式錯誤消滅在採樣層

工具呼叫要求模型輸出符合 JSON Schema 的參數。靠提示詞祈求模型「請輸出合法 JSON」是 2023 年的做法;2024

年起, 主流 API 提供 **constrained decoding**(受限解碼): 在每一步採樣前, 依據由 schema 編譯成的文法 (或正則、上下文無關文法) 遮蔽所有會導致非法輸出的 token, 使輸出**構造性地**合法。

- OpenAI 的 structured outputs(`strict: true`) 自 2024 年 8 月 GA, 對函數參數與回應格式均可強制 schema。
- Anthropic 於 2025 年 11 月推出 structured outputs beta(`structured-outputs-2025-11-13`), 同時覆蓋回應格式 (`output_format`) 與 strict tool use(保證工具參數 schema 合法)。
- GPT-5 世代進一步支持以上下文無關文法/正則約束純文字工具輸出 (custom tools + CFG)。

工程結論: 凡 schema 可精確表達的格式要求, 應交給 constrained decoding, 而不是提示詞; 提示詞負責語義正確性, 採樣層負責語法正確性。這是「把可判定的問題從機率域移回確定域」這一 harness 總原則 (第十一章) 的第一個實例。

2.4 推理模型與思考預算

2024 年末以來, 經強化學習訓練的推理模型 (reasoning models,OpenAI o 系列、Claude 的 extended thinking、Gemini 的 thinking 模式、DeepSeek-R1 等) 在生成最終答案前先生成思維鏈 (chain of thought)。對 harness 的影響有四:

1. **思考 token 計費且佔用上下文**。思考內容通常計入輸出費用; 在 Anthropic 的實現中,thinking token 亦計入上下文預算。長時程代理的上下文管理必須把它納入核算。
2. **思考預算成為可調參數**。Claude Opus 4.5 引入 effort 參數,Opus 4.6 擴至四檔 (low/medium/high/max) 並引入 adaptive thinking(由模型自行決定何時深思);Gemini 3 有對應的 thinking level。實測數據顯示其槓桿極大:Opus 4.5 在 medium effort 下即可追平 Sonnet 4.5 的最佳 SWE-bench 成績, 同時輸出 token 減少 76%。effort 路由 (按任務難度分配思考預算) 因此成為成本工程的一級手段 (第二十章)。

3. **推理狀態的跨輪傳遞成為協議問題**。OpenAI 的 Responses API 以 reasoning items 在工具呼叫輪之間保存推理狀態;Gemini 3 要求開發者在後續函數呼叫輪中原樣回傳加密的 thought signatures, 否則 API 直接報錯。harness 若自行管理對話歷史 (而非用供應商的會話原語), 必須正確搬運這些不透明塊——這是 2026 年新出現的一類 harness bug 來源。
4. **思維鏈是受控的 scratchpad, 不是可靠的自白**。Anthropic 的多代理研究系統文章將 extended thinking 描述為「可控的思考稿紙」——它可用於引導模型規劃, 但把它當作模型真實決策過程的忠實記錄用於審計, 在對齊研究上已被證明不可靠。可觀測性設計 (第十八章) 應以行動與終態為準, 思維鏈僅作輔助信號。

2.5 從單次推理到代理迴圈

至此可以精確描述 harness 的最內層——代理迴圈。以 Anthropic 的表述, 迴圈的每一輪是三步: **收集上下文** → **採取行動** → **驗證工作** (gather context → take action →

verify work)。以偽碼表示其最小完整形態:

```
def agent_loop(task: str, tools: list[Tool], policy: Policy) ->
↪  Result:
    context = Context(system=SYSTEM_PROMPT, tools=tools)
    context.append(user(task))
    while True:
        response = model.generate(context)          # 一次推理
        if response.has_tool_calls():
            for call in response.tool_calls:
                decision = policy.check(call)        # 權限層
                if decision is Deny:
                    context.append(tool_error(call,
↪  decision.reason))
                    continue
                if decision is AskHuman:
                    await human_approval(call)
                result = execute(call)               # 執行環境
                result = truncate(result, MAX_TOOL_TOKENS)  # 上下
↪  文防護
                context.append(tool_result(call, result))
        else:
            if not verifier.accepts(response):       # 驗證層
                context.append(user(verifier.feedback()))
                continue
```

```
            return response.final()
        context = manage(context)                    # compaction
↪  / 卸載
```

這二十餘行偽碼中,`policy.check`、`truncate`、`verifier.accepts`、`manage` 四個掛載點, 分別對應第八、七、十一、七章的主題;`tools` 的設計是第五章的主題; 整個迴圈的持久化與恢復是第十四、十五章的主題。換言之, **本書的結構就是這個迴圈的展開**。

值得注意的是迴圈中兩個容易被低估的細節:

- **工具結果截斷** (`truncate`) 不是可有可無的防護: 單次 `cat` 一個大檔案或一次失控的測試輸出, 足以吞噬整個上下文窗口。Claude Code 預設在 25,000 token 處截斷工具回應; 任何自建 harness 都需要等價機制。
- **錯誤也要回饋給模型** (`tool_error` 分支)。Anthropic 的生產經驗是:「告訴代理某個工具正在失效、讓它自行調適, 效果出奇地好」。吞掉錯誤、靜默重試, 反而剝奪了模型繞道的機會。MCP 2025-11-25 修訂版甚至把「輸入驗證錯誤應作為工具執行錯誤返回、以

便模型自我修正」寫進了規格 (SEP-1303)。

2.6 嵌入與語義檢索: 輔助機制的定位

嵌入 (embedding) 模型把文字映射為高維向量, 使語義相近的內容在向量空間中鄰近, 支撐語義檢索 (RAG 的檢索側)。在 2023 至 2024 年的技術敘事中, 向量檢索幾乎是「讓模型接觸外部知識」的同義詞; 但在 2025 年之後的代理式 harness 中, 它的地位被重估。

Anthropic 在 Claude Agent SDK 的設計闡述中給出的排序是:**agentic search 優先,semantic search 按需**。agentic search 指讓模型用 grep、glob、tail 這類確定性工具自行探索檔案系統——結果精確、無需維護嵌入索引與分塊管線、天然與檔案系統的權限及時效一致; 語義檢索更快, 但精度較低, 且引入分塊 (chunking)、嵌入更新、索引一致性等一整套需要維護的基礎設施,「只在 agentic search 不足時使用」。第七章將把這場「檢索之爭」放在 just-in-time context 的框架下完整展開。

這個重估本身是削減式哲學的一個實例: 向量檢索管線是為「模型無法自行高效探索大語料」這一假設建造的; 當模

型加迴圈加 grep 已經能勝任大部分程式庫導航時, 該假設部分失效, 相應基礎設施的維護成本便需要重新論證。

2.7 本章小結

- Token 是成本、上下文預算與若干模型缺陷的共同計量單位; 中文的 token 密度劣勢與 JSON 的轉義開銷是 harness 設計的實際輸入。
- KV cache 與 prompt caching 要求上下文採取「穩定前綴 + 增量尾部」佈局; 長上下文任務的累計成本呈平方增長, 構成 compaction 與隔離技術的經濟學動機。
- RoPE 外推的位置分辨率損失、n^2 注意力攤薄與訓練分佈偏短, 共同構成 context rot 的機制解釋 (第三章)。
- 非確定性經代理迴圈乘法放大; 可判定的格式問題應以 constrained decoding 在採樣層根治。
- 推理模型引入思考預算 (effort)、跨輪推理狀態 (reasoning items / thought signatures) 與新的成本槓桿。
- 代理迴圈「收集上下文 → 行動 → 驗證」的四個掛載

點 (權限、截斷、驗證、上下文管理) 構成 harness 的骨架, 也構成本書的章節結構。

本章參考文獻

1. Anthropic, “Effective Context Engineering for AI Agents”, 2025-09-29. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents
2. Anthropic, “Building Agents with the Claude Agent SDK”, 2025-09-29. https://claude.com/blog/building-agents-with-the-claude-agent-sdk
3. Anthropic, “Claude Opus 4.5” 發布說明 (effort 參數與 token 效率數據), 2025-11-24. https://www.anthropic.com/news/claude-opus-4-5
4. Anthropic, “Claude Opus 4.6” 發布說明 (adaptive thinking、compaction API), 2026-02-05. https://www.anthropic.com/news/claude-opus-4-6
5. Google, Gemini API 文檔:Function calling 與

Thought signatures. https://ai.google.dev/gemini-api/docs/function-calling

6. OpenAI, Structured Outputs 文檔與 Responses API 遷移指南. https://platform.openai.com/docs/guides/migrate-to-responses
7. Anthropic, Structured Outputs(beta) 文檔. https://platform.claude.com/docs/en/build-with-claude/structured-outputs
8. Anthropic, “How We Built Our Multi-Agent Research System”, 2025-06-13. https://www.anthropic.com/engineering/built-multi-agent-research-system # 第三章上下文窗口的物理學:Context Rot 與注意力預算

供應商的規格表寫着「支持 200k / 1M token 上下文」, 而生產系統的工程師很快發現: **標稱長度與有效長度是兩回事**。模型在長上下文下的能力退化——業界稱為 context rot(上下文腐化)——不是個別產品的缺陷, 而是當前 Transformer 架構下可測量、可預測、跨模型一致的系統性現象。本章先陳列實證, 再給出機制解釋, 最後把它轉化為 harness 設計的第一約束: **注意力預算 (attention**

budget)。

3.1 實證一: 位置效應——Lost in the Middle

Liu 等人 2023 年 7 月的〈Lost in the Middle: How Language Models Use Long Contexts〉(arXiv:2307.03172, 後刊於 TACL) 是這個領域的奠基實驗。在多文檔問答與鍵值檢索任務中, 將關鍵資訊放置在上下文的不同位置, 測得的準確率呈 **U 形曲線**: 資訊位於開頭 (primacy) 與結尾 (recency) 時表現最好, 位於中段時顯著下降——某些配置下, 中段位置的表現甚至低於完全不提供該文檔的閉卷基線。且這一效應在明確標榜長上下文的模型上依然存在。

對 harness 的直接推論: 上下文佈局是有位置語義的。系統提示與核心指令放最前, 當前任務與最新狀態放最後, 大宗參考材料的中段位置最不可靠——「把關鍵約束重複於結尾」這類看似粗糙的技巧, 有堅實的實證基礎。

3.2 實證二: 長度效應的精細解剖——Chroma 的 Context Rot 報告

Chroma 於 2025 年 7 月 14 日發表的技術報告〈Context Rot: How Increasing Input Tokens Impacts LLM Performance〉對 18 個模型 (涵蓋 Claude 3.5/3.7/4 系列、GPT-4o/4.1/o3、Gemini 2.0/2.5、Qwen3 系列) 做了迄今最系統的長度—性能實驗。其結論值得逐條記錄, 因為每一條都對應一個 harness 設計決策:

1. **針—題語義距離決定退化速度**。經典的 needle-in-a-haystack(NIAH) 測試用字面匹配的針, 是最容易的情形。當針與問題只有語義關聯而無詞面重疊時, 性能隨長度退化的速度顯著加快。推論:NIAH 滿分不能證明長上下文可用; 你的任務裏「針」與「題」的語義距離越遠, 有效上下文越短。
2. **干擾項 (distractors) 的破壞力隨長度放大**。與針相似但不正確的內容, 在短上下文中影響輕微, 在長上下文中導致非均勻的性能崩塌, 且模型會具體地幻覺出干擾項內容。推論: 向上下文堆砌「可能相關」的

材料不是無害的保險, 而是主動注入干擾。

3. **結構的反直覺效應**。邏輯連貫的 haystack 反而比隨機打亂的更難——模型在連貫文本中似乎更容易被敘事流帶走而忽略檢索目標。
4. **退化與任務難度無關地普遍存在**。即使是「重複輸出一段詞序列」這種零難度任務, 長上下文下也會失敗。
5. **聚焦優於全量**。在 LongMemEval 上, 提供聚焦的相關摘錄的表現顯著優於提供完整對話歷史。這一條直接論證了第七章的核心主張: 檢索—聚焦架構優於「把一切塞進窗口」。

報告還記錄了模型間的行為差異:Claude 系列在不確定時傾向保守棄答 (abstain),GPT 系列更傾向自信地幻覺。這類「失敗風格」差異影響 harness 的錯誤處理設計——面對保守型模型要防止過早放棄, 面對自信型模型要加強驗證。

3.3 實證三: 去除詞面匹配——NoLiMa

Adobe Research 的 NoLiMa(arXiv:2502.05167,ICML 2025) 把「針—題無詞面重疊」推到極致: 檢索必須經過一步潛在的語義聯想 (latent hop)。結果: 在 32k token 長度下, 受測 12 個模型中有 11 個跌破其短上下文基線的 50%。即使是推理模型, 在 NoLiMa-Hard 上同樣顯著退化。這宣判了「我們通過了 NIAH, 所以長上下文沒問題」這一常見論證的死刑。

配合 NVIDIA 的 RULER(多針、變量追蹤、聚合、多跳問答的合成套件; 核心結論是多數模型的有效上下文遠短於標稱值) 與 LongBench v2(503 道 8k–2M 詞的困難選擇題; 人類專家在 15 分鐘限時下僅得 53.7%, 發布時最佳模型約 57.7%),2024 至 2025 年的長上下文評估文獻收斂出一致圖景: **上下文長度是一種會退化的資源, 退化速率取決於任務的語義結構**。

3.4 機制解釋: 為什麼會腐化

第二章已鋪墊了三個機制, 這裏正式匯總:

1. **注意力攤薄**。 n 個 token 之間有 n^2 對注意力關係;softmax 歸一化意味着注意力是零和的——上下文越長, 每個 token 平均分得的注意力質量越低。無關內容不是中性填充, 它直接稀釋有效信號。
2. **訓練分佈偏短**。訓練語料中長序列佔比小, 模型用於處理全上下文依賴的參數容量相對不足; 長上下文能力部分依賴後期的長度外推訓練, 泛化不完全。
3. **位置編碼外推損耗**。 RoPE 內插/縮放讓模型「見過」的位置密度被拉伸, 位置分辨率下降, 遠距離的精確定位變得模糊。

這三個機制的共同含義是:context rot 不會被下一代模型完全消除, 只會被推遲和減輕。Claude Opus 4.6 在 MRCR v2 長上下文基準上得 76%(對照 Sonnet 4.5 的 18.5%) 顯示了世代進步的幅度; 但注意力的零和性質是架構性的。**在可預見的架構範式內, 上下文管理不是可選優化, 而是永久的工程約束**。——這也是為什麼本書把 context rot 稱為「物理學」: 你可以工程化地繞開它, 但不能假裝它不存在。

3.5 注意力預算: 把約束轉化為設計語言

Anthropic 的〈Effective Context Engineering〉提出了本領域最有用的心智模型: 把上下文窗口視為一種**有限且邊際收益遞減的預算**。每個進入窗口的 token 都在消費這個預算;harness 的職責是使每一單位預算的邊際資訊價值最大化。

先看清預算表上有什麼。一個生產代理在第 t 輪推理時, 上下文窗口的典型構成:

條目	典型量級	特性
系統提示	1–5k tokens	穩定前綴, 可緩存
工具定義	5–100k+ tokens	穩定前綴; 未經治理時是最大的隱性開銷
指令檔 (CLAUDE.md / AGENTS.md 等)	0.5–5k tokens	穩定前綴
對話與行動歷史	隨輪次線性增長	增量尾部

條目	典型量級	特性
工具結果 (檔案內容、測試輸出等)	波動極大, 單次可達數萬 tokens	主要的預算殺手
思考 tokens	依 effort 設定	計入預算
當前用戶訊息	小	尾部

Anthropic 披露的一個內部數據點極具代表性: 在治理之前, 其自家某個內部配置的工具定義佔用了 134k token 的上下文——尚未開始任何對話, 預算已耗去大半。這解釋了為什麼 2025 年末開始,「工具定義的按需載入」(tool search / deferred loading, 第五、九章) 成為各家 API 的一級功能: 實測顯示可削減約 85% 的相關開銷 (72k → 8.7k token), 並同時提升工具選用準確率 (Opus 4 在 MCP 評估上從 49% 升至 74%——更少的干擾項, 更好的選擇, 與 3.2 節的干擾項發現完全一致)。

退化梯度 (degrade gradient)

工程上有用的另一個描述是: 上下文不是在某個閾值瞬間「爆掉」, 而是沿一個梯度劣化——高相關內容逐漸被噪音混入, 早期資訊被 compaction 壓縮走樣, 細節逐步失真, 直到模型開始引用不存在的變量、重複已完成的步驟、或忘記最初的約束。生產系統中觀察到的「代理越跑越笨」現象, 多數可歸因於這個梯度, 而非模型能力的隨機波動。

由此得出 harness 上下文治理的三條操作性原則 (第六、七章展開為具體技術):

1. **預算導向**: 對上下文的每一類佔用建立核算; 工具定義、指令檔、工具結果各有專門的削減手段 (deferred loading、精簡指令、截斷與卸載)。
2. **位置導向**: 穩定內容前置以吃緩存, 關鍵約束靠近尾部以吃 recency, 大宗材料不放中段。
3. **梯度監控**: 長時程任務中把「上下文健康度」(佔用率、compaction 次數、工具結果佔比) 作為可觀測性指標 (第十八章), 在退化造成行為異常之前介入。

3.6 「上下文焦慮」:模型對預算的自我感知

2025 年末出現了一個微妙的新現象: 模型不僅受上下文限制, 還會**對限制產生行為反應**。Anthropic 在長時程 harness 實驗中觀察到,Claude Opus 4.5 在感知到上下文窗口將盡時, 傾向於倉促收尾——放棄尚未完成的子任務、提前宣告勝利。他們將其命名為 **context anxiety(上下文焦慮)**。

這個現象的 harness 含義是雙重的。短期, 它催生了補償機制:2025 年 11 月的長時程 harness 以「每會話只做一個 feature + 頻繁上下文重置」來規避模型在深上下文區間的行為異常。長期, 它成為削減式哲學的教科書案例:Opus 4.6 顯著緩解了 context anxiety,Anthropic 隨即刪除了 sprint 合約等補償組件。**針對模型行為癖性的 harness 組件, 壽命以模型世代為單位**。設計時就應標注「此組件補償的是哪個模型的哪個行為, 升級時重新評估」。

與此同時, 供應商也把緩解手段下沉到了 API 層:Anthropic 於 2025 年 9 月推出 context edit-

ing(自動清除陳舊工具結果) 與 memory tool, 實測在 100 輪網頁搜索任務中, 僅 context editing 就削減 84% 的 token 消耗、並使原本因上下文耗盡而失敗的任務得以完成;Opus 4.6(2026 年 2 月) 進一步提供服務端的 compaction API。第七章將系統比較這些機制。

3.7 對「百萬 token 上下文」的正確態度

Opus 4.6 與 Gemini 系列已提供 1M token 級的上下文 (beta)。面對這類規格,harness 工程師應持的立場:

1. **長窗口擴大的是可行域, 不是最優域**。 1M 窗口讓「整個程式庫一次放入」在技術上可行, 但 3.2–3.4 節的全部證據表明, 聚焦的上下文在準確率上優於全量灌入; 且超過 200k 的部分往往按更高費率計費 (Opus 4.6 為 $10/$37.50 每百萬 token), 成本梯度本身就在懲罰粗放使用。
2. **長窗口改變的是 compaction 的頻率, 不是其必要性**。對於以天計的長時程任務 (第十四章), 任何有限窗口都會被填滿; 窗口加長只是把管理壓力後移。
3. **長窗口是給「不可分解的整體性任務」的**——需要同

時看到全部素材的交叉引用分析、大規模重構的一致性檢查。對可分解任務, 子代理隔離加摘要回傳 (第十三章) 幾乎總是更優。

3.8 本章小結

- Context rot 是跨模型一致的實證現象: 位置 U 形曲線 (Lost in the Middle)、長度—語義距離交互效應、干擾項放大、結構反直覺效應 (Chroma 18 模型報告)、無詞面匹配下的斷崖退化 (NoLiMa:32k 處 12 模型中 11 個跌破短上下文基線之半)。
- 機制: 注意力零和攤薄、訓練分佈偏短、位置編碼外推損耗——決定了 context rot 可被減輕而不可被消除。
- 注意力預算是 harness 的第一設計約束: 對上下文的每類佔用建立核算, 以 deferred loading、截斷、卸載、compaction 治理; 佈局遵循「穩定前綴吃緩存、關鍵約束吃 recency」。
- Context anxiety 表明模型會對預算產生行為反應; 針對此類癖性的補償組件應標注預期壽命, 隨模型升級重新評估。

- 百萬 token 窗口不改變聚焦優於全量的結論, 只改變管理壓力的位置。

本章參考文獻

1. Liu et al., “Lost in the Middle: How Language Models Use Long Contexts”, arXiv:2307.03172, 2023-07.
2. Chroma, “Context Rot: How Increasing Input Tokens Impacts LLM Performance”, 2025-07-14. https://www.trychroma.com/research/context-rot
3. Adobe Research, “NoLiMa: Long-Context Evaluation Beyond Literal Matching”, arXiv:2502.05167, ICML 2025.
4. NVIDIA, “RULER: What’s the Real Context Size of Your Long-Context Language Models?”, arXiv:2404.06654, 2024-04.
5. LongBench v2, arXiv:2412.15204, 2024-12. https://longbench2.github.io/
6. Anthropic, “Effective Context Engineering for

AI Agents”, 2025-09-29.

7. Anthropic, “Introducing Advanced Tool Use”(tool search 數據), 2025-11-24. https://www.anthropic.com/engineering/advanced-tool-use
8. Anthropic, “Managing Context on the Claude Developer Platform”(context editing / memory tool 數據), 2025-09-29. https://www.anthropic.com/news/context-management
9. Anthropic, “Harness Design for Long-Running Application Development”(context anxiety), 2026-03-24.
10. Mei et al., “A Survey of Context Engineering for Large Language Models”, arXiv:2507.13334, 2025-07. # 第四章代理的分類學:Workflow、Agent 與 Augmented LLM

「Agent」可能是 2024 至 2026 年間被濫用得最嚴重的技術詞彙。本章的任務是恢復它的精確性: 給出可操作的定義邊界, 建立從確定性自動化到自主代理的完整光譜, 並確立「先簡後繁」的架構決策紀律。這些區分不是學究式

的——選錯抽象層級是代理專案失敗的頭號架構原因。

4.1 基準定義

沿用 Anthropic〈Building Effective Agents〉的定義, 這是目前業界接受度最高的一組:

- **Workflow(工作流)**:LLM 與工具透過**預先定義的程式碼路徑**被編排的系統。控制流由開發者寫死; 模型只在路徑中的特定節點提供智能。
- **Agent(代理)**:LLM **動態地指導自身的過程與工具使用**、對如何完成任務保持控制權的系統。控制流由模型在運行時決定。

兩者的上游是 **augmented LLM(增強型 LLM)**: 單次推理的模型, 配備檢索、工具與記憶。它是兩類系統共同的原子構件。

判別標準可以壓縮為一個問題:「**下一步做什麼」是誰決定的?** 若答案寫在你的程式碼裏 (哪怕分支再多), 它是 workflow; 若答案來自模型在運行時的輸出, 它是 agent。

這條邊界在工程上的意義是**方差的位置**。Workflow 把模

型的非確定性封閉在節點內部, 節點之間的轉移是確定的, 系統整體方差有界;agent 讓非確定性進入控制流本身, 方差隨步數乘法累積 (第二章 2.3 節)。因此: 方差可控性要求高、流程可預先枚舉的場景用 workflow; 路徑無法預先枚舉、需要運行時探索的場景才值得付出 agent 的方差代價。Anthropic 的原文告誡至今有效:「先從簡單開始, 只在確有必要時增加複雜度」;LangChain 生態的文檔亦持相同立場。

4.2 自動化光譜

把相鄰概念放在同一光譜上, 混亂即消失:

層級	「下一步」的決定者	非確定性	例子
Automation(自動化)	完全寫死	無	cron 定時備份
Rule-based workflow	人類預寫的規則分支	無	金額 >1 萬轉人工審批

層級	「下一步」的決定者	非確定性	例子
LLM-in-workflow	路徑寫死, 節點含模型	節點內	郵件分類後定向轉發
Agent	模型運行時決定	控制流級	修復一個未知成因的 bug
Multi-agent	多個模型分工決定	系統級	研究系統的 orchestrator-workers

注意兩個常見的分類錯誤。其一, 把「用了 LLM」等同於「是 agent」——n8n、Zapier 類自動化工具中嵌入的 AI 節點, 只是 workflow 中的一個智能環節, 不構成 agent, 儘管這條界線隨這類產品引入完整代理節點而持續模糊。其二, 把「有多個 prompt」等同於「是 multi-agent」——prompt chaining 是單一控制流下的多次推理, 仍是 workflow。

4.3 五種 Workflow 模式

〈Building Effective Agents〉確立的五種模式已成為業界的標準詞彙。此處給出每種模式的結構、適用條件與失效邊界; 第十二章將配以完整程式碼實現。

1. Prompt chaining(提示鏈)。任務分解為固定順序的步驟, 每步輸出經過可選的程式化關卡 (gate) 後成為下一步輸入。適用: 可清晰分解、每步可驗證的任務 (生成 → 校驗 → 潤色)。關鍵工程點:gate 必須是程式化判定 (schema 驗證、規則檢查), 否則鏈條只是在傳遞未經過濾的方差。

2. Routing(路由)。先以一次分類決定輸入屬於哪一類, 再分派到專門的下游提示、模型或流程。適用: 輸入存在明確可分的類別, 且各類的最優處理方式不同。關鍵工程點: 路由輸出必須 schema 化 (structured output 強制枚舉值), 絕不接受自由文字; 類別應設 fallback 分支。路由也是成本工程的入口——簡單類別派給小模型 (第二十章的 model routing)。

3. Parallelization(並行化)。兩個變體:**sectioning**(獨

立子任務並行執行後聚合) 與 **voting**(同一任務並行執行多次, 以多數決或評分聚合提升信度)。適用: 子任務彼此獨立 (sectioning), 或單次執行方差高而正確性可投票判定 (voting)。失效邊界: 子任務之間存在隱含依賴時,sectioning 產生互相矛盾的輸出——這正是第十三章多代理衝突問題的單代理版本。

4. Orchestrator-workers(協調者—工作者)。中央 LLM 在運行時動態分解任務、派發給 worker、綜合結果。與 parallelization 的區別在於: 子任務**由輸入決定而非預先定義**。這是 workflow 光譜上最接近 agent 的模式, 也是多代理系統的原型。適用: 分解方式依輸入而變的任務 (多檔案修改、開放式研究)。

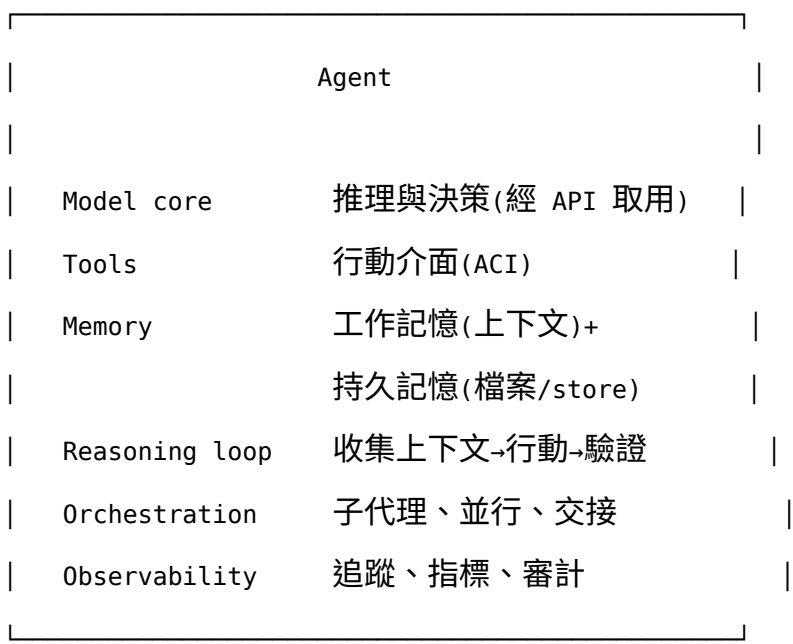
5. Evaluator-optimizer(評估者—優化者)。一個模型生成, 另一個模型依明確標準評估並給出回饋, 循環迭代。適用: 存在清晰評估標準、且迭代確實能改進的任務 (文檔打磨、翻譯、程式碼評審意見生成)。失效邊界: 評估標準模糊時, 循環退化為兩個模型互相恭維——第十一章「自我評估問題」的雛形。

五種模式是可組合的: 生產系統常見「routing 之後接

prompt chaining、其中一步是 parallel voting」的複合結構。組合的紀律只有一條: **每增加一層, 都要能指出它對可靠性或能力的具體貢獻, 否則刪掉**。

4.4 Agent 的解剖

當任務確實需要運行時自主性, 系統進入 agent 範疇。一個生產級 agent 的完整解剖 (對照第一章的 harness 七組件, 此處從系統組成的視角重述):



↑ 全部運行於 `harness` 提供的執行環境之內

這個六要素解剖 (model core + tools + memory + reasoning loop + orchestration + observability) 不是任何一家的官方定義, 而是對 Anthropic、LangGraph、OpenAI 三方文獻的工程綜合; 它的價值在於作為架構評審的檢查清單——six 個要素中任何一個缺位, 系統就會以可預測的方式失敗 (無 observability 則不可調試, 無 verification 迴圈則不可信, 無持久記憶則不可長時程)。

VS Code 對代理迴圈的表述——Understand → Act → Validate——與 Anthropic 的 gather context → take action → verify work 同構, 可視為同一結構的兩種措辭。迴圈的三步分別由 harness 的三類組件支撐: 上下文管理 (第六、七章)、工具與執行環境 (第五、八章)、驗證機制 (第十一章)。

4.5 決策框架: 何時用哪一層

實務中的選型可以編纂為一個決策序列:

1. **流程可以完整畫成流程圖嗎?** 可以 → 不要用 agent。

用 workflow 模式組合, 把模型限制在節點內。收益:方差有界、成本可預算、調試確定性。

2. **失敗的代價是否高於探索的價值?** 是 → 即使流程不可完全枚舉, 也應以 workflow 為骨架、agent 為局部 (受限自主性:agent 只在沙箱化的子問題內自主)。
3. **任務是否需要運行時探索、且環境能提供行動的真實回饋?** 是 →agent 是正確選擇。但 Anthropic 的限定條件必須同時滿足:「用於難以或不可能預測所需步數的開放性問題」, 且環境中存在可供驗證的回饋信號 (測試、編譯器、瀏覽器狀態)。無回饋信號的環境中,agent 的迭代只是漫遊。
4. **單一上下文窗口是否裝得下任務所需的全部視野?** 裝不下 → 考慮子代理隔離 (第十三章) 或長時程機制 (第十四章)。裝得下 → 留在單代理, 這幾乎總是更優 (多代理的 15 倍 token 成本與協調失敗模式, 見第十三章)。

一個輔助性的經驗判據來自 Claude Code 最佳實踐文檔對規劃的建議, 可移植到架構選型:「如果你能用一句話描述 diff, 就跳過規劃」——同理, 如果你能用一張流程圖描述系統, 就跳過 agent。

4.6 反模式目錄

本章以四個高頻架構反模式作結, 每一個都在生產環境中被反覆觀察到:

反模式一: 用 agent 實現確定性管線。把一條本可寫死的 ETL 流程交給代理「自主完成」, 收穫的是每天不同的執行路徑與無法復現的失敗。診斷特徵: 代理的工具呼叫序列在 95% 的運行中完全相同——那 5% 的自由度只在製造事故。處方: 把 95% 固化為 workflow,5% 的真實分支交給 routing。

反模式二: 框架黑箱依賴。 Anthropic 在 2024 年已警告: 框架的便利「容易掩蓋底層的 prompt 與 response, 使調試更難」, 且「對底層機制的錯誤假設是客戶錯誤的常見來源」。處方: 無論使用何種框架, 團隊必須有能力 dump 任意一輪推理的完整原始輸入輸出; 選型時把「透明度」列為一級標準 (第十五章的框架比較將以此為軸之一)。

反模式三: 過早多代理化。「一個 PM agent、一個架構師 agent、一個工程師 agent 開會」的擬人化架構, 在 2025 年的失敗研究 (MAST 分類學, 第十三章) 中被證明主要產

生協調失敗而非分工收益。多代理的唯一有效動機是上下文隔離、並行探索與工具集特化, 而不是模仿人類組織結構。

反模式四: 無驗證的自主性。給了代理自主權而沒有給環境驗證信號——沒有測試、沒有 lint、沒有可斷言的終態。結果是代理宣稱完成而無人能廉價地判定真偽, 系統的信任成本回落到逐行人工審查, 自主性的收益歸零。第十一章的中心論點在此預告: **自主性的上限由驗證能力決定, 而非由模型能力決定**。

4.7 本章小結

- Workflow 與 agent 的判別標準:「下一步做什麼」由程式碼決定還是由模型運行時決定; 本質區別是非確定性方差進入控制流的位置。
- 五種 workflow 模式 (chaining、routing、parallelization、orchestrator-workers、evaluator-optimizer) 構成標準詞彙, 可組合; 每加一層需論證其貢獻。
- Agent 六要素解剖:model core、tools、memory、

reasoning loop、orchestration、observability——缺一即以可預測方式失敗。

- 決策序列: 能畫流程圖 →workflow; 失敗代價高 → 受限自主; 需真實探索且有回饋信號 →agent; 單窗口裝不下 → 隔離或長時程機制。
- 四個反模式:agent 化的確定性管線、框架黑箱、過早多代理化、無驗證的自主性。

本章參考文獻

1. Anthropic, “Building Effective Agents”, 2024-12-19.
2. LangChain, “LangGraph 文檔:Workflows and Agents”. https://docs.langchain.com/
3. Anthropic, “Building Agents with the Claude Agent SDK”, 2025-09-29.
4. Cemri et al., “Why Do Multi-Agent LLM Systems Fail?”(MAST), arXiv:2503.13657, NeurIPS 2025.
5. Claude Code Best Practices. https://code.claude.com/docs/en/best-practices # 第二部 Harness

的組件 {.part}

第五章工具介面設計:Agent-Computer Interface 的工程學

工具是模型與世界之間唯一的行動通道。Anthropic 對工具的定義精確地捕捉了其特殊性: 工具是「一種新型軟件, 它反映確定性系統與非確定性代理之間的一紙契約」。傳統 API 的消費者是程式——行為固定、嚴格按文檔呼叫; 工具的消費者是模型——會誤讀、會遺漏、會被描述中的每一個詞影響。因此工具介面設計 (agent-computer interface, ACI) 是一門獨立的工程學, 其投入產出比被反覆證明高於提示工程:Anthropic 在其 SWE-bench 工作中花在工具上的優化時間超過整體提示, 而 2024 年那次 33%→49% 的躍升主要來自工具層。

5.1 工具的解剖

一個工具在 API 層面由四部分構成:

```
{
  "name": "search_contacts",
  "description": "Search the user's contact list by name, email,
  ↪  or company.
    Returns up to `limit` matches ranked by relevance.
    Use this instead of list_contacts when looking for a specific
↪   person.",
  "input_schema": {
    "type": "object",
    "properties": {
      "query":  { "type": "string", "description": "Name, email
      ↪  fragment, or company" },
      "limit":  { "type": "integer", "default": 10, "maximum": 50
      ↪  }
    },
    "required": ["query"]
  }
}
```

四個組成部分各自的設計負載:

- **name**: 模型選擇工具時的第一信號。命名空間前綴 (`asana_search`、`asana_projects_search`) 幫助模型在多服務環境中定位;Anthropic 的實驗甚至顯示前綴式與後綴式命名空間會產生可測量的性能差異。MCP 2025-11-25 修訂版專門發布了工具命名指引 (SEP-986)。
- **description**: 實質上是嵌入在每次請求中的微型提示。應寫給「一個對你系統一無所知的新同事」: 何時用、何時不用、與相鄰工具的邊界、參數的隱含約定。Anthropic 的直接證據: 僅打磨工具描述, 就「戲劇性地降低了錯誤率」並貢獻了當時的 SWE-bench SOTA。
- **input_schema**: 配合 strict mode / constrained decoding(第二章 2.3 節) 後,schema 即語法契約。schema 本身也是文檔: 枚舉值、default、範圍約束都在向模型傳遞語義。
- **輸出格式** (不在 schema 中, 但同樣是契約): 見 5.4 節。

5.2 選擇建造哪些工具: 對抗「API 鏡像」反射

最常見的工具集設計錯誤, 是把現有 REST API 一比一鏡像為工具——每個 endpoint 一個工具, 得到一個數十個工具的扁平清單。這在三個層面失敗: 工具定義吞噬上下文預算 (第三章); 功能重疊製造模糊決策點 (Anthropic 明確警告「臃腫的工具集覆蓋過多功能, 導致模糊的決策邊界」); 而模型被迫在多次呼叫之間人肉搬運中間結果, 每一次搬運都消耗 token 並引入抄寫錯誤。

正確的方法論是**面向工作流建造, 而非面向資源建造**:

- 識別代理的高頻工作流, 為每條工作流建造一個「打通到底」的工具。Anthropic 的例子: 與其提供 `list_events` + `create_event` 讓模型自行組合, 不如提供一個 `schedule_event`(內部完成找空檔與建立); 與其 `list_contacts` 讓模型翻頁篩選, 不如 `search_contacts`。
- 合併返回: 一個 `get_customer_context` 一次性返回客戶的關鍵摘要, 優於讓模型串行呼叫五個 getter。

- 每個工具做一件在工作流層面完整的事。判準: 模型是否經常需要把此工具的原始輸出直接餵給另一個工具? 若是, 兩者可能應該合併, 或應該下沉到程式碼執行層 (5.6 節)。

5.3 防呆設計 (Poka-yoke): 以介面消滅錯誤類別

ACI 的核心方法論繼承自製造業的防呆思想: **與其教育使用者不犯錯, 不如讓錯誤在介面上不可能發生**。已被文獻記錄的實例:

- **強制絕對路徑**。Anthropic 的 SWE-bench 代理曾因相對路徑產生一整類定位錯誤; 把工具改為要求絕對路徑後,「這類錯誤被完全消滅」。
- **選擇對模型自然的格式**。要求模型輸出帶行號差異的嚴格 diff 格式, 是把簿記負擔壓在最不擅長簿記的組件上; 字串替換式編輯 (提供唯一匹配的 old_string/new_string) 把負擔轉移給確定性代碼。同理,「接近自然網絡文本」的格式優於重度轉義的格式。

- **參數化而非拼接**。任何把模型輸出直接插入 shell 字串、SQL 字串的介面, 都同時是注入漏洞與錯誤放大器; 工具內部應使用參數化介面 (execve 參數陣列、prepared statement)。
- **以 enum 收窄自由度**。凡取值可枚舉的參數 (輸出格式、排序方式、目標環境) 一律 enum 化, 把拼寫錯誤從運行時移到 schema 驗證層。
- **危險操作的介面隔離**。刪除、覆寫、對外發送等不可逆操作應是獨立工具 (而非通用工具的一個參數), 使權限層 (第八章) 能按工具粒度施加審批;MCP 的工具註解 (read-only / destructive hints,2025-03-26 修訂版) 正是為此設計的協議級支持。

5.4 工具輸出: 上下文預算的第一戰場

工具結果通常是上下文中最大的變動開銷 (第三章的預算表), 其設計原則:

只返回高信號資訊。 Anthropic 的 Slack 工具實驗: 同一查詢,`concise` 格式 72 token,`detailed` 格式 206 token——三倍之差, 而多數場景 concise 已足夠。實踐是給工具一

個 `response_format` enum, 讓模型 (或 harness 策略) 按需選擇詳略。

語義識別符優於不透明識別符。返回 `"channel": "#eng-infra"` 而非 `"channel_id": "C04XJ81QW"`: 語義識別符讓模型少一次解引用呼叫, 且在後續推理中不易抄錯 (第二章的 tokenization 論據)。需要精確 ID 做後續操作時, 兩者並返。

分頁、過濾、截斷是工具的義務, 不是模型的美德。工具應內建 `limit` 與過濾參數, 並在返回中明示截斷情況 (「顯示前 20/1,342 條, 使用 offset 取得更多」)。harness 層再設一道硬截斷兜底 (Claude Code 的 25,000 token 預設)。

錯誤訊息是指令, 不是 stack trace。「query 過寬, 返回 12,000 條; 請加入 date_range 或更具體的關鍵詞」引導模型自我修正; 裸 traceback 只是消耗預算。MCP 2025-11-25 將「輸入驗證錯誤以工具執行錯誤形式返回以便模型修正」寫入規格, 同一思想的協議化。

5.5 規模化: 當工具有幾十個到上萬個

工具數量增長帶來雙重壓力: 定義佔用 (靜態) 與選擇準確率 (干擾項效應, 第三章 3.2 節)。2025 年末形成的分層解法:

第一層:deferred loading + 工具檢索。 Anthropic 的 Tool Search Tool(2025-11,beta `advanced-tool-use-2025-11-20`): 工具標記 `defer_loading: true` 後不佔上下文, 模型透過一個服務端檢索工具 (regex 或 BM25 變體) 按需發現,API 將 `tool_reference` 展開為完整定義; 支持至一萬個延遲工具。實測: 典型多 MCP 服務器配置的工具定義開銷從 72k 降至 8.7k token(約 85%), 同時工具選用準確率上升 (Opus 4 於 MCP 評估 49%→74%)。這證實了一個重要判斷: **裁剪工具清單不僅省錢, 而且直接提升行為品質**——干擾項少了。

第二層: 工具使用示例。對參數複雜的工具,`input_examples`(同一 beta) 提供少樣本示例, 複雜參數處理準確率 72%→90%。示例是 schema 無法表達的慣例 (嵌套結構的典型形態、可選欄位的組合規律) 的正確載體。

第三層: 程式碼執行作為聚合介面。見下節。

5.6 終極形態: 把工具呈現為程式碼 API

2025 年 11 月,Anthropic 的〈Code Execution with MCP〉指出了直接工具呼叫模式的兩個規模化病理: 所有定義前置載入, 以及**每個中間結果都必須穿過模型**——從 Google Drive 取一份兩小時會議記錄轉存 Salesforce,transcript 在上下文中來回一趟, 額外燒掉約五萬 token。

解法是把 MCP 服務器呈現為**檔案系統中的程式碼 API**(如 `servers/google-drive/getDocument.ts`), 讓模型寫程式碼呼叫它們: 中間結果留在沙箱變量裏, 迴圈/條件/重試在程式碼層完成而非推理層, 只有最終摘要回到上下文。示範場景的 token 從 150,000 降到 2,000(98.7%)。附帶收益:progressive disclosure(模型按需讀取工具定義檔)、隱私 (PII 可在沙箱內全程 tokenize)、以及技能沉澱 (模型「把自己的程式碼持久化為可重用函數」)。

二十天後, 此模式產品化為 API 級的 **Programmatic Tool Calling**:Claude 在代碼執行容器內以

Python 呼叫開發者的客戶端工具 (`allowed_callers: ["code_execution_20250825"]`), 生產工作負載上的實測收益較示範場景保守但仍顯著: 複雜研究任務 token 減少 37%(43,588→27,297), 多項基準準確率同步上升。

工程結論: 工具數量少、單次呼叫即完成的場景, 直接工具呼叫仍是最簡架構; 工具多、存在資料密集的多步編排時, 「工具即程式碼 API + 沙箱執行」是方向。其前提——一個安全的代碼執行沙箱——是第八章的主題。

5.7 工具的評估與自我改進

工具是軟件, 軟件需要測試。Anthropic 的工具評估方法論:

1. **以真實多步任務為單位**, 而非單工具單元測試。強任務示例:「與 Jane 安排會議, 附上上次規劃會議的記錄, 並訂會議室」——覆蓋搜索、讀取、建立三類工具的協同。
2. **度量四元組**: 準確率之外, 同時記錄運行時間、工具呼叫次數、token 消耗、錯誤率。兩套工具集可能準確率相同而成本相差數倍。

3. **讓代理改進自己的工具**。把失敗任務的完整 transcript 交給 Claude Code 分析並重寫工具描述與介面——agent 優化後的工具在保留集上勝過人類手寫版本; 其多代理研究系統中, 一個工具測試代理透過重寫工具描述, 使後續任務完成時間縮短 40%。這是「自我改進 harness」目前最可靠的落地形式: 改進的是確定性 artifact(工具定義), 而非模型本身, 每一次改進都可審查、可版本化、可回滾。

5.8 本章小結

- 工具是確定性系統與非確定性代理之間的契約;name/description/schema/輸出四部分各有設計負載,description 的打磨有直接的基準分數證據。
- 面向工作流建造工具, 對抗 API 鏡像反射; 功能重疊即干擾項。
- Poka-yoke: 絕對路徑、字串替換編輯、參數化執行、enum 收窄、危險操作介面隔離——以介面設計消滅整類錯誤。
- 輸出設計: 高信號、語義識別符、內建分頁過濾、指令式錯誤訊息;response_format 分級。

- 規模化三層:deferred loading + 檢索 (≈85% 定義開銷削減且準確率上升)、使用示例 (72%→90%)、程式碼執行聚合 (示範 98.7%、生產 37% 的 token 削減)。
- 工具需要以真實任務、四元組指標評估;「代理重寫工具」是當前最穩健的自我改進迴路。

本章參考文獻

1. Anthropic, “Writing Effective Tools for Agents — With Agents”, 2025-09-11. https://www.anthropic.com/engineering/writing-tools-for-agents
2. Anthropic, “Building Effective Agents” 附錄二 (ACI), 2024-12-19.
3. Anthropic, “Raising the bar on SWE-bench Verified”, 2024-10-30.
4. Anthropic, “Introducing Advanced Tool Use on the Claude Developer Platform”, 2025-11-24. https://www.anthropic.com/engineering/advanced-tool-use

5. Anthropic, “Code Execution with MCP: Building More Efficient Agents”, 2025-11-04. https://www.anthropic.com/engineering/code-execution-with-mcp
6. Anthropic, Programmatic Tool Calling 文檔. https://platform.claude.com/docs/en/agents-and-tools/tool-use/programmatic-tool-calling
7. Model Context Protocol, 2025-11-25 changelog(SEP-986 工具命名、SEP-1303 驗證錯誤). https://modelcontextprotocol.io/specification/2025-11-25/changelog
8. Anthropic, “How We Built Our Multi-Agent Research System”(工具測試代理 40% 數據), 2025-06-13. # 第六章上下文工程 I: 系統提示、指令檔與記憶層級

上下文工程的定義, 依 Anthropic 的表述, 是「在 LLM 推理過程中策展與維護最優 token 集合的策略」。與一次性的提示工程不同,「上下文工程是迭代的, 每次決定向模型傳遞什麼時, 策展階段都會重新發生」。本章處理上下文中**相對穩定的部分**——系統提示、指令檔與記憶體系; 下一章

處理**動態的部分**——檢索、compaction 與卸載。

6.1 系統提示: 高度的藝術

系統提示的設計難點不在措辭而在**抽象高度 (altitude)**。Anthropic 描述了兩個失敗極端與一個目標區間:

- **過低**: 把業務邏輯寫成脆弱的偽 if-else——「若用戶提到退款且金額大於 X 則……否則若……」。這實質上是在用自然語言寫程式, 享受不到程式的可測試性, 也享受不到模型的泛化能力; 每個新情況都需要人工補一條分支, 提示無限膨脹。
- **過高**:「你是一個樂於助人的專業助手, 請提供高品質回覆」——空洞指導, 假定模型與你共享未言明的標準, 實際輸出方差極大。
- **目標區間 (Goldilocks zone)**:「足夠具體以有效引導行為, 又足夠靈活以提供強啟發式」。給出決策的**原則與判準**, 而非枚舉每個分支; 給出**具體的錨定示例**, 而非抽象形容詞。

結構上的最佳實踐已趨於收斂: 以區段組織 (背景、指令、工具使用政策、輸出格式), 使用 Markdown 標題或 XML

標籤定界; 先以最小提示 + 最強模型建立基線, 再針對實測失敗模式增補——而不是預防性地堆砌想像中的規則。少樣本示例遵循同一紀律:「策展多樣、典範的示例集」, 勿把邊界情況全部塞入;「對 LLM 而言, 示例是一圖勝千言的那個圖」。

系統提示還受第二章緩存約束的直接支配: 它是最典型的穩定前綴, 任何運行時變量 (時間戳、用戶名) 都應移到訊息尾部或以佔位符延遲注入, 避免擊穿 prompt cache。

6.2 指令檔: 版本化的專案上下文

2025 至 2026 年,「把團隊規範寫成 repository 內的 agent 可讀檔案」從各家私有慣例收斂為開放格式。這一層的本質是: **把原本存在於資深工程師腦中的隱性知識, 轉化為可版本控制、可評審、每次會話自動載入的顯性 artifact**。

格式生態

- **AGENTS.md**: 開放標準,「代理的 README」。刻意的極簡設計: 純 Markdown、無必填欄位、無 schema; 放在 repo 根目錄,monorepo 中可嵌套

(就近優先); 用戶會話中的明確指示優先於檔案。由 OpenAI Codex、Amp、Google Jules、Cursor、Factory 等共同發起 (2025 年 8 月整合),2025 年 12 月隨 AAIF(Agentic AI Foundation,Linux Foundation 旗下) 成立而移交中立治理; 公開統計超過六萬個開源 repo 採用,Codex、Gemini CLI、Copilot/VS Code、Cursor、Aider、Junie、Zed 等原生支持。

- **CLAUDE.md**:Claude Code 的原生指令檔, 層級化載入 (企業/用戶/專案/子目錄), 支持 `@path` 匯入。與 AGENTS.md 功能同構;Claude Code 對 AGENTS.md 的原生支持長期是社群高熱請求, 實務慣例是在 CLAUDE.md 中 `@AGENTS.md` 匯入或以符號連結橋接, 使兩個生態共用一份事實。
- 其他:Cursor 的 Rules、Gemini CLI 的 GEMINI.md(可配置改讀 AGENTS.md)、Cline 的 `.clinerules`——同一概念的方言。

格式之爭不重要, 重要的是內容紀律。

內容紀律: 寫什麼、不寫什麼

兩份最佳實踐文獻——Claude Code 官方文檔與 OpenAI 的 harness engineering 文章——給出了高度一致的準則:

寫入: 模型無法從程式碼自行推斷的資訊。建置與測試指令 (尤其是非標準的)、架構決策及其理由、偏離語言慣例的專案約定、環境設置的陷阱、repo 禮儀 (commit 規範、分支策略)。

不寫入: 模型讀程式碼就能知道的東西 (檔案結構的逐條描述、顯而易見的慣例)、大而全的風格指南 (交給 linter 機械執行, 見第十一章)、以及「以防萬一」的冗長背景。Claude Code 文檔的警告直白:「臃腫的 CLAUDE.md 會導致 Claude 忽略你真正的指令」——指令檔同樣受注意力預算與干擾項效應 (第三章) 支配。

形態:OpenAI 團隊的實踐提供了規模上限的參考——服務百萬行程式碼專案的 AGENTS.md 約一百行, 定位是「地圖, 而非千頁說明書」: 它指向結構化的 `docs/` 目錄樹, 細節放在按需讀取的文檔中 (下一章 just-in-time 的靜態版

本)。Hashimoto 的實踐提供了增量紀律的參考:「檔案中的每一行都對應一個曾實際發生的糟糕代理行為」——指令檔是失敗驅動的沉澱物, 不是預防性的百科全書。

指令檔應納入 code review 流程: 它是影響所有代理會話的高槓桿 artifact, 一行壞指令的破壞面等同一個壞的全域配置。

6.3 記憶層級: 工作記憶之外

上下文窗口是代理的工作記憶——快、貴、易失。跨會話的知識需要外部記憶體系。2026 年的生產實踐可歸納為四層:

層	載體	生命週期	典型內容
會話內筆記	上下文中的結構化記錄 / scratchpad 檔案	單會話	當前計劃、待辦、中間結論

層	載體	生命週期	典型內容
專案記憶	指令檔、`docs/`、progress 檔案	隨 repo 永續	規範、架構、任務進度
用戶/組織記憶	harness 管理的記憶檔案目錄	跨專案持久	偏好、環境事實
外部記憶系統	向量庫 / 圖式記憶 (Letta、Mem0、A-MEM 等)	應用定義	對話歷史蒸餾、實體關係

結構化筆記 (structured note-taking) 是其中投入產出比最高的技術。原理: 讓代理把跨越 compaction 邊界必須存活的狀態, 主動寫入上下文之外的持久載體 (NOTES.md、todo 清單、progress 檔案), 需要時再讀回。Anthropic 的 Pokémon 實驗展示了其威力: 代理在

數千步遊戲中維持精確計數 (「過去 1,234 步我在 1 號道路訓練, 皮卡丘距目標 10 級還差 2 級」), 在上下文重置後讀回筆記無縫續接。長時程 harness(第十四章) 的 `claude-progress.txt` 與 feature JSON 是同一技術的工程化形態。

API 級記憶工具。Anthropic 的 memory tool(2025 年 9 月,`memory_20250818`) 把這一模式標準化: 模型獲得對一個**客戶端檔案目錄**的增刪改查能力, 目錄由開發者的基礎設施持有, 跨會話持久。設計要點在於它是純客戶端的——沒有服務端魔法, 開發者完全控制儲存、審計與清理。與 context editing 合用, 在內部代理式搜索評估上帶來 39% 的提升。

外部記憶系統的學術與產品譜系——MemGPT 的 OS 式虛擬上下文管理 (其商業化為 Letta)、A-MEM 的 Zettelkasten 式動態鏈接筆記、Mem0 的生產記憶層 (自報在 LOCOMO 上較基線準確率 +26%、p95 延遲 -90%)、Letta 的 sleep-time compute(在互動間隙離線重組記憶, 以約五分之一的測試時算力達到相近準確率)——共同的架構教訓是: **記憶的價值取決於寫入時的蒸餾品質, 而非讀出時的檢索技巧**。把原始對話全量寫入再靠檢索補救, 是把

context rot 搬到了記憶庫裏。

記憶的反面: 污染與投毒

記憶是跨會話的狀態, 也因此是跨會話的攻擊面與錯誤放大器。工程上必須設防的兩類問題:

1. **自我污染**: 代理把錯誤結論寫入記憶, 後續會話將其當作事實。對策: 記憶寫入應偏好「可驗證的事實與決策記錄」而非「推斷」; 定期以來源重新驗證; 給記憶條目附時間戳與來源。
2. **記憶投毒**: 攻擊者透過間接注入 (第十九章) 誘導代理寫入惡意指令, 實現跨會話持久化攻擊。Anthropic 的 containment 文章明確把「持久記憶投毒 (CLAUDE.md、掛載工作區)」列為未來風險。對策: 記憶檔案納入與指令檔同級的審查; 高權限會話不自動載入低信任來源寫入的記憶。

6.4 佈局學: 把穩定性變成結構

綜合本章與第二、三章, 生產 harness 的上下文佈局準則:

```
┌ 穩定前綴(prompt cache 覆蓋區)──────────────┐
```

```
| 1. 系統提示(無運行時變量)                          |
| 2. 工具定義(經 deferred loading 治理後的核心集)|
| 3. 指令檔(CLAUDE.md / AGENTS.md,精簡)          |
├ 增量區 ──────────────────────────────┤
| 4. 對話與行動歷史(僅追加;compaction 有策略地 |
|    重寫,接受一次性緩存失效)                    |
| 5. 工具結果(截斷、卸載後的殘餘)                 |
├ 尾部(recency 覆蓋區)──────────────────┤
| 6. 當前任務狀態 / 關鍵約束的重申                  |
| 7. 用戶最新訊息                                 |
└──────────────────────────────────────────┘
```

三條結構化紀律:

1. **前綴凍結**: 區段 1–3 的任何變動都應被視為「部署」——有版本、有 diff、有評審, 因為它同時影響行為與緩存經濟。
2. **歷史僅追加**: 區段 4 在正常運行中只追加不修改;compaction 是显式的、有日誌的例外。
3. **約束重申**: 長會話中,harness 可在尾部週期性重申任務目標與硬約束 (利用 recency 效應對抗中段迷失), 這是廉價而有效的 context rot 對沖。

6.5 本章小結

- 系統提示的核心變量是抽象高度: 避免偽程式與空洞指導, 以原則 + 判準 + 典範示例佔據 Goldilocks 區間; 失敗驅動地增補, 而非預防性堆砌。
- 指令檔 (AGENTS.md/CLAUDE.md) 是版本化的隱性知識; 寫模型推斷不出的東西, 保持「地圖」形態 (百行級, 指向 docs 樹), 每一行對應一個真實失敗; 納入 code review。
- 記憶四層: 會話筆記、專案記憶、用戶/組織記憶、外部記憶系統; 結構化筆記是最高性價比技術; 記憶價值取決於寫入蒸餾品質。
- 記憶需設防: 自我污染與跨會話投毒; 記憶檔案應與指令檔同級審查。
- 佈局: 穩定前綴吃緩存、歷史僅追加、關鍵約束尾部重申。

本章參考文獻

1. Anthropic, “Effective Context Engineering for AI Agents”, 2025-09-29.

2. agents.md(AGENTS.md 開放格式). https://agents.md
3. Claude Code Best Practices(CLAUDE.md 準則). https://code.claude.com/docs/en/best-practices
4. OpenAI, “Harness Engineering”(百行 AGENTS.md 與 docs 樹), 2026-02-11.
5. Mitchell Hashimoto, “My AI Adoption Journey”, 2026-02-05.
6. Anthropic, “Managing Context on the Claude Developer Platform”(memory tool), 2025-09-29.
7. Packer et al., “MemGPT: Towards LLMs as Operating Systems”, arXiv:2310.08560;Letta, “Sleep-time Compute”, arXiv:2504.13171.
8. Mem0, arXiv:2504.19413;A-MEM, arXiv:2502.12110.
9. Anthropic, “How We Contain Claude Across Products”(記憶投毒風險), 2026-05-25. # 第七章 上下文工程 II: 檢索、Compaction 與上下文卸載

穩定層 (第六章) 決定代理「一開始知道什麼」; 動態層決

定代理「在漫長的任務中還能繼續知道什麼」。本章覆蓋上下文的三大動態操作: 按需引入 (just-in-time retrieval)、有損壓縮 (compaction) 與向外轉移 (offloading), 以及它們在 2025–2026 年的 API 化進程。

7.1 Just-in-Time: 從預載入到按需載入

傳統 RAG 管線的心智模型是「推理前準備好一切」: 嵌入、檢索、把 top-k 片段預先注入提示。代理式 harness 的心智模型與之相反: **上下文中只保留輕量識別符——檔案路徑、查詢語句、URL、資料庫鍵——實際資料在需要的那一刻經工具載入**。

Anthropic 將此類比於人類的外部記憶: 我們不背誦整個檔案櫃, 我們記得哪個抽屜放什麼。並指出一個容易被忽視的事實: **元資料本身就是信號**——檔案名、目錄層級、時間戳在資料被讀取之前就在傳遞語義 (`tests/legacy/` 之下的檔案自帶「過時」標記)。

Agentic search 對 semantic search

第二章已預告這場重估, 此處給出完整的決策框架:

維度	Agentic search(grep/glob/讀檔)	Semantic search(嵌入檢索)
精度	精確匹配, 零假陽性幻覺	近似匹配, 存在假陽性/假陰性
延遲	多輪工具呼叫, 較慢	單次檢索, 較快
基礎設施	無需索引, 零維護	嵌入管線、分塊策略、索引一致性、權限同步
時效	永遠是當前狀態	索引滯後窗口
適用	結構化語料 (程式庫、配置、文檔樹)	大規模非結構化語料、跨語言語義匹配

Anthropic 的立場 (Claude Agent SDK 設計闡述) 是明確的預設優先級:agentic search 為預設,semantic search 是在延遲敏感、語料非結構化、或 agentic search 實測不足時的升級選項。Claude Code 本身是混合策略的範

本:「CLAUDE.md 被樸素地前置載入, 而 glob 與 grep 原語讓它按需檢索檔案」——穩定層預載, 動態層 JIT。

工程上還有第三條路: **為代理準備易於導航的靜態結構**。OpenAI 的 docs 樹 (AGENTS.md 作為地圖指向分層文檔) 本質是為 agentic search 優化的資訊架構——不建索引, 而是把語料整理成模型一兩跳就能命中的形狀。這往往是三者中投入產出比最高的。

7.2 Compaction: 有損壓縮的工程學

任何有限窗口終將被長任務填滿, 屆時只有三個選項: 失敗、截斷、或壓縮。Compaction 指以模型 (或專用摘要模型) 將既有歷史重寫為緊湊摘要, 再以摘要重新初始化上下文續跑。

分層的 compaction 策略

實踐中 compaction 不是單一操作, 而是一個激進程度遞增的策略族:

1. **工具結果清理 (tool result clearing)**: 最輕、最安全的一層。歷史中的舊工具結果 (讀過的檔案內容、

跑過的測試輸出) 在後續輪次中價值急降——保留「讀了什麼、結論如何」, 清掉原始內容。Anthropic 將其稱為「最安全的輕觸式 compaction」, 並已 API 化 (context editing 的 `clear_tool_uses_20250919` 策略)。實測:100 輪搜索任務中單此一項削減 84% token 消耗。

2. **階段性摘要**: 把已完成階段的推理與行動壓縮為結構化摘要 (目標、關鍵決策、產出 artifact 的位置), 保留最近若干輪的完整細節。調參原則 (Anthropic): **先最大化 recall(不丟關鍵資訊), 再提升 precision(削冗餘)**——丟失一個關鍵約束的代價遠高於多留幾百個 token。
3. **上下文重置 (context reset)**: 完全清空, 僅以結構化交接檔案 (progress 檔、feature 清單、git log) 冷啟動新會話。這已不是壓縮而是**換代**——依賴的是外部記憶 (第六章) 而非摘要品質。Anthropic 在 2026 年 3 月的文章中明確區分 reset 與 compaction, 並指出長時程系統中兩者的適用場景不同:compaction 適合任務內的連續性維持,reset 適合階段邊界 (此時乾淨上下文的價值高於歷史細節)。

Compaction 的風險面

- **緩存失效**:compaction 重寫歷史, 擊穿其後全部 KV cache(第二章); 觸發時機應與緩存經濟共同優化——在自然邊界 (階段完成、會話切換) 觸發優於在任意水位觸發。
- **資訊丟失的靜默性**: 被壓掉的細節不會報錯, 只會在數十輪後表現為代理「忘記」某個約束。對策: 硬約束不進 compaction 流 (放指令檔或尾部重申區);compaction 摘要納入 trace(第十八章) 供事後歸因。
- **摘要偏置**: 摘要模型傾向保留敘事主線、丟棄反例與未決問題。提示中應顯式要求保留「未解決的問題、已排除的方案及其原因」——後者恰是防止代理重走死路的關鍵資訊。

API 化進程

Compaction 在 2025–2026 年迅速從應用層下沉:Claude Code 內建自動 compaction(接近水位觸發,`/compact` 手動觸發);Claude Agent SDK 與 LangChain 1.0 的 summarization middleware 提供框架級實現;Opus 4.6 提供

服務端 compaction API(beta), 由供應商在接近窗口上限時自動壓縮舊歷史;Microsoft Agent Framework Harness 把 context compaction 列為 GA 功能。應用層自建 compaction 的理由正在縮小, 但**控制壓縮什麼、保留什麼的策略仍屬 harness 職責**——服務端 compaction 不知道你的任務中哪些是不可丟的硬約束。

7.3 卸載 (Offloading): 把狀態移出窗口

Compaction 是把資訊壓小; 卸載是把資訊移走, 只留指標。三種形態:

1. **工具結果卸載**: 大體積工具輸出直接落盤 (`/tmp/test_output_1423.log`), 上下文中只留路徑與摘要行; 需要時代理以 grep/tail 部分讀回。Carlini 的 C 編譯器專案的實踐表述:「保持測試輸出簡潔, 把細節記錄到檔案, 以便 Claude 需要時能找到」。
2. **狀態卸載**: 計劃、待辦、中間結論寫入結構化檔案 (第六章的 structured note-taking)——上下文重置後的生存基礎。
3. **計算卸載**: 資料密集操作整體移入代碼執行層 (第五

章 5.6 節), 中間結果根本不進入上下文。從一萬行的表格中過濾出五行, 過濾發生在沙箱, 窗口只見五行。

三種卸載共同構成一個原則: **上下文窗口是 CPU 快取, 不是主存**。設計問題從「怎樣把東西放進去」轉為「怎樣讓多數東西不必進去」。

7.4 子代理作為上下文架構

子代理 (subagent) 通常被理解為並行化手段, 但其更本質的價值是**上下文隔離**: 一個子代理帶着乾淨的窗口深入探索 (讀幾十個檔案、跑幾輪搜索), 向父代理返回一段濃縮摘要——Anthropic 的經驗值是典型 1,000–2,000 token。父代理的窗口只承擔摘要, 探索過程的數萬 token 被結構性地隔絕在外。

這使子代理成為 compaction 的**前瞻性替代**: 與其讓探索污染主上下文再事後壓縮, 不如讓污染從未發生。Claude Code 的 Task 工具、各框架的 subagent 原語 (第十五章) 都支持這一用法。其代價與完整的多代理權衡 (共享與隔離的張力、token 經濟) 留待第十三章; 此處只確立一條使用即回報的規則: **凡「大量讀、少量出」的偵察型任務, 一**

律子代理化。

7.5 動態層的整體設計: 一個參考策略

把本章機制組裝為一個可直接採用的預設策略 (以 200k 窗口為例):

- **0–60% 佔用**: 正常運行。工具結果進上下文前先截斷 (單次上限如 25k); 超過閾值的輸出自動落盤改留指標。
- **60–80%**: 啟動工具結果清理 (保留最近 N 輪); 偵察型子任務強制子代理化。
- **80–90%**: 階段性 compaction(recall 優先的摘要提示); 同時把當前計劃與未決事項強制寫入狀態檔——為最壞情況準備 reset 基礎。
- **>90% 或階段邊界**: 上下文重置, 以狀態檔 +git 歷史冷啟動。
- **全程**: 上下文佔用率、compaction 次數、單輪工具結果均值進入監控 (第十八章); 異常增長是工具設計缺陷 (第五章) 的信號, 應修工具而非調水位。

這套水位線本身是削減式哲學的對象: 服務端 com-

paction 與更大窗口會不斷把閾值後移, 但「截斷—清理—壓縮—重置」的升級序列結構是穩定的。

7.6 本章小結

- JIT 原則: 上下文保留識別符, 資料按需經工具載入; 元資料本身是信號;agentic search 為預設,semantic search 按實測需要引入; 為代理優化資訊架構常勝過建索引。
- Compaction 策略族按激進度分層: 工具結果清理 (單項可省 84%)→ 階段摘要 (recall 先於 precision)→ 上下文重置 (依賴外部記憶而非摘要); 風險是緩存失效、靜默丟失與摘要偏置。
- 卸載三形態 (工具結果、狀態、計算) 把窗口從主存降格為快取;「大量讀少量出」的任務一律子代理隔離 (摘要回傳 1–2k token)。
- 動態層已快速 API 化 (context editing、服務端 compaction、框架 middleware), 但「什麼不可丟」的策略永遠是 harness 職責。

本章參考文獻

1. Anthropic, “Effective Context Engineering for AI Agents”, 2025-09-29.
2. Anthropic, “Building Agents with the Claude Agent SDK”, 2025-09-29.
3. Anthropic, “Managing Context on the Claude Developer Platform”, 2025-09-29.
4. Anthropic, “Harness Design for Long-Running Application Development”(reset 與 compaction 之辨), 2026-03-24.
5. Anthropic, “Claude Opus 4.6”(服務端 compaction API), 2026-02-05.
6. Nicholas Carlini / Anthropic, “Building a C Compiler with a Team of Parallel Claudes”(輸出落盤實踐), 2026-02-05. https://www.anthropic.com/engineering/building-c-compiler
7. OpenAI, “Harness Engineering”(docs 樹作為檢索結構), 2026-02-11. # 第八章執行環境: 檔案系統、Shell 與沙箱

「給你的代理一台電腦, 讓它像人類一樣工作。」——這是 Claude Agent SDK 的核心設計原則, 也是 2025 年之後 harness 設計最重要的收斂: 與其為每種能力建造專用工具, 不如給代理一個真實的計算環境 (檔案系統、shell、程式語言運行時), 讓通用計算成為能力的母體。但一台真實的電腦同時是一個真實的攻擊面。本章處理這對矛盾: 如何給予最大的計算自由, 同時維持可證明的安全邊界。

8.1 檔案系統與 Shell: 通用性的來源

代理環境以檔案系統為中心的理由, 在前幾章已多處出現, 此處匯總為系統性論證:

1. **檔案系統是天然的外部記憶** (第六章): 筆記、進度、計劃以檔案持久化, 天然跨會話、天然可 git 版本化。
2. **檔案系統是上下文工程的載體** (第七章): 卸載的目的地、JIT 檢索的對象、「工具即程式碼 API」的呈現層。
3. **Shell 是萬能適配器**: 任何有 CLI 的既有軟件 (git、docker、psql、ffmpeg、雲廠商 CLI) 零成本地成為代理能力;Unix 管道的組合性直接繼承。

4. **程式碼生成是最高密度的行動形式**:「程式碼是精確、可組合、無限可重用的」; 一段腳本可以表達一千次工具呼叫才能完成的操作。

工具層級因此形成一個金字塔: 頂端是少數精心設計的專用工具 (高頻工作流, 佔據上下文中的顯著位置、直接塑造行為); 中部是 bash 與腳本 (通用膠水); 底部是 MCP 連接的外部服務 (第九章)。專用工具的數量應被刻意壓低——每個新工具都在消耗注意力預算並與 bash 的通用性重疊; 判準是「此工作流是否高頻且易錯到值得一個一級介面」。

8.2 威脅模型: 為什麼沙箱不是可選項

給代理 shell, 等於給每一段能影響模型輸出的文字——用戶輸入、讀到的檔案、抓到的網頁、工具返回的資料——一條通往任意程式碼執行的潛在路徑。第十九章將完整展開 prompt injection 的攻防; 本章先確立執行層的結論: **模型層防禦不充分, 必須存在一個模型無法說服的確定性邊界**。

Anthropic 的 containment 文章提供了一個殘酷的實證

錨點: 在一次內部釣魚測試中, 面對直接注入,25 次嘗試中 24 次成功實現資料外洩。同文的另一個發現同樣重要:「最薄弱的一層是你自己建造的那一層」——自建的代理、允許清單與過濾器的失效率, 顯著高於 gVisor、seccomp 這類久經考驗的系統級隔離原語。工程含義: 安全邊界應盡可能建立在作業系統與虛擬化層, 而非應用邏輯層。

8.3 隔離技術譜系

2026 年生產系統使用的隔離原語, 按邊界強度排列:

普通容器 (Docker/OCI):namespace + cgroup 隔離, 共享宿主內核。啟動快、生態成熟; 但內核攻擊面完整暴露, 單一內核漏洞即可逃逸。適合信任度較高的工作負載或作為多層防禦的一層。

gVisor: 用戶態內核 (Sentry) 攔截並重新實現 syscall, 應用不直接觸達宿主內核。Google 內部大規模使用;Anthropic 的 claude.ai 代碼執行環境採用 gVisor 容器 (短暫、每會話銷毀)。代價是 syscall 密集型負載的性能損耗。

microVM(Firecracker):KVM 硬件虛擬化, 每實例獨立內核, 約 125ms 級啟動。源自 AWS Lambda; 是 E2B、Fly Machines 等代理沙箱平台的基底。邊界最強的實用選項。

WebAssembly: 能力式安全模型,syscall 面極窄。適合插件式工具的嵌入式隔離; 作為完整代理 shell 環境尚非主流。

OS 級原語:Linux 的 namespaces/bubblewrap、seccomp-bpf(syscall 過濾)、Landlock(非特權檔案系統沙箱);macOS 的 Seatbelt(`sandbox-exec`)。無需容器基礎設施, 適合本地開發工具——Claude Code 的本地沙箱 (開源為 `anthropic-experimental/sandbox-runtime`) 在 Linux 用 bubblewrap、macOS 用 Seatbelt;OpenAI Codex CLI 同樣以 Seatbelt/bubblewrap+seccomp 實現其沙箱模式, 並為 Windows 專門構建了原生沙箱。

完整 VM: 最強隔離。Anthropic 的 Cowork(桌面代理產品) 使用完整密封 VM(macOS 上為 Apple Virtualization framework), 並且把代理迴圈移到 VM **外部**運行——只有程式碼執行在 VM 內, 模型與憑證在外。這個「腦在箱外」

的架構是 8.5 節 Managed Agents 模式的桌面版。

8.4 兩個必要邊界: 檔案系統與網絡

Anthropic 的沙箱文章給出了本領域最重要的一條架構定理:

> 「有效的沙箱同時需要檔案系統隔離與網絡隔離。沒有網絡隔離, 被攻陷的代理可以外洩 SSH 金鑰等敏感檔案; 沒有檔案系統隔離, 被攻陷的代理可以輕易逃逸。」

檔案系統隔離: 寫入範圍收窄到工作目錄 (及顯式掛載), 敏感路徑 (`~/.ssh`、憑證存放處、shell 配置) 不可讀。Cowork 的掛載模式細分為 read-only、read-write、read-write-no-delete 三檔——最後一檔對「代理誤刪用戶檔案」這一高頻事故是精確的介面級防呆。

網絡隔離: 預設拒絕出站, 經**沙箱外的代理伺服器**放行域名允許清單。代理程序在沙箱內、執行策略在沙箱外, 這個位置關係是關鍵——策略引擎必須在代理 (以及被注入的代理) 無法觸及的信任域。

允許清單本身需要威脅建模:containment 文章記錄了「經批准域名外洩」的紅隊發現——「允許清單上任何域名可達的每一個功能, 都是攻擊面」(例如經允許的 GitHub 域名, 向攻擊者控制的 repo 推送資料)。因此高敏感環境的清單應精確到路徑與方法, 而不僅是域名。

這套「filesystem scoping + default-deny egress + 憑證隔離」已成為跨廠商共識:Codex cloud 的任務容器在 setup 階段聯網、代理階段預設斷網;E2B/Modal/Daytona 等沙箱平台的差異化已不在隔離強度 (人人都有硬邊界), 而在快照/分叉語義、冷啟動延遲與出口策略控制粒度。

實效數據: 沙箱化使 Claude Code 的權限提示減少 84%——安全與自主性在此不是取捨, 而是相互成全: **邊界越硬, 邊界內的自由度越可以放開**。這是本章的第二條定理。

8.5 憑證架構: 秘密不進沙箱

沙箱內的一切都應被假定可能被注入攻擊接管, 因此**長期憑證絕不進入沙箱**。生產模式:

- **代理側持有, 呼叫時注入**:API 金鑰保存在沙箱外的 harness 層; 工具呼叫由 harness 代執行, 沙箱內的程式碼只見結果不見金鑰。Managed Agents 的表述:「憑證從不進入沙箱」。
- **短期化與窄化**: 必須進入環境的憑證使用短生命 token、最小 scope;OAuth 流程 (第九章 MCP 授權) 在 harness 層完成,resource indicator 防止 token 被挪用於非目標服務。
- **秘密掃描兜底**:commit 前的 secret scanning(Gitleaks 類) 作為 CI 關卡, 攔截模型「好心」寫死金鑰的行為。

8.6 「腦與手分離」: 執行環境的架構終局

Anthropic 的 Managed Agents 文章 (2026 年 4 月) 把執行環境的架構抽象推到目前的終點, 其三元模型值得完整轉述:

- **Brain(腦)**: 模型呼叫與 harness 邏輯, 作為無狀態服務運行於容器**之外**; 持有憑證與策略。

- **Hands(手)**: 按需創建的沙箱與工具,「牛群而非寵物」——可隨時銷毀重建, 不承載不可再生狀態。
- **Session(會話)**: 持久的 append-only 事件日誌, 系統的唯一事實來源; 恢復即 `wake(sessionId)` 重放。

這個分離的直接收益是運維性的 (消除前置容器供應使首 token 延遲 p50 降約 60%、p95 降逾 90%; 任何組件可獨立升級與水平擴展), 但其深層意義是安全與狀態模型的: 腦在信任域、手在非信任域、狀態在兩者之外的持久層——三者各自失效都不損害另兩者。第十四、十五章的長時程與持久化設計將建立在這個模型上。

對照 2023–2024 年「代理 = 一個長跑的進程, 狀態在記憶體裏, 憑證在環境變量裏」的原始形態, 腦手分離是執行環境三年演化的濃縮: **計算可拋棄、狀態可重放、秘密不落地**。

8.7 本地開發環境的務實配置

不是每個場景都需要 microVM。按風險分級的務實建議:

- **個人開發、可信 repo**:OS 級沙箱 (srt/Seatbelt/bubblewrap

出站允許清單 + 敏感路徑遮蔽, 已淘汰「逐條批准每個指令」的體驗 (approval fatigue 本身就是安全風險——用戶對 93% 的權限請求無腦放行, 見第十一章 auto mode 的討論)。

- **CI 與自動化 (headless)**: 容器 + 預設斷網 + 顯式放行; 工作區即拋。
- **多租戶/接觸不可信輸入的生產代理**:gVisor 或 microVM, 腦手分離, 憑證外置。
- **接觸用戶本機資料的桌面代理**: 完整 VM + 掛載模式細分 + 代理迴圈外置 (Cowork 模式)。

8.8 本章小結

- 「給代理一台電腦」: 檔案系統 (記憶/上下文載體)+ shell(萬能適配器)+ 程式碼生成 (最高密度行動) 構成通用性母體; 專用工具保持少而精。
- 威脅模型結論: 必須存在模型說服不了的確定性邊界; 邊界應建於 OS/虛擬化層——「最薄弱的一層是你自己建的那一層」。
- 隔離譜系: 容器 < gVisor < microVM ≤ 完整 VM; 本地工具用 OS 級原語 (bub-

blewrap/Seatbelt/seccomp/Landlock)。

- 兩個必要邊界定理: 檔案系統隔離 + 網絡隔離缺一不可;default-deny egress 由沙箱外代理執行; 允許清單按「域名上每個可達功能都是攻擊面」做威脅建模。
- 邊界越硬, 箱內自由越大 (權限提示 -84%); 憑證不進沙箱, 短期化、窄化、掃描兜底。
- 架構終局是腦手分離: 計算可拋棄、狀態可重放 (append-only session)、秘密不落地。

本章參考文獻

1. Anthropic, “Building Agents with the Claude Agent SDK”, 2025-09-29.
2. Anthropic, “Beyond Permission Prompts: Making Claude Code More Secure and Autonomous”, 2025-10-20. https://www.anthropic.com/engineering/claude-code-sandboxing
3. anthropic-experimental/sandbox-runtime. https://github.com/anthropic-experimental/sandbox-runtime

4. Anthropic, “How We Contain Claude Across Products”, 2026-05-25. https://www.anthropic.com/engineering/how-we-contain-claude
5. Anthropic, “Scaling Managed Agents: Decoupling the Brain from the Hands”, 2026-04-08.
6. OpenAI, “Building a Safe, Effective Sandbox to Enable Codex on Windows”. https://openai.com/index/building-codex-windows-sandbox/
7. OpenAI Codex 安全與審批文檔. https://learn.chatgpt.com/docs/agent-approvals-security
8. AWS, Firecracker microVM;Google, gVisor 專案文檔. # 第九章 Model Context Protocol: 工具連接的標準層

在 MCP 之前, 每個 harness 與每個外部服務之間的整合都是一次性工程:M 個代理應用 × N 個服務 = M×N 個適配器。Model Context Protocol 於 2024 年 11 月由 Anthropic 開源, 以一層標準協議把問題降維為 M+N; 到 2026 年, 它已是被全行業採用、由中立基金會治理、經歷五個修訂版打磨的事實標準——各方索引合計約十萬個服務器實現, 官方 SDK 月下載量近億。本章從架構、演化

史、安全與工程實務四個角度解剖這個協議, 並特別關注其修訂史所折射的分散式系統教訓。

9.1 架構:Host、Client、Server 與原語

MCP 是基於 JSON-RPC 2.0 的協議, 三個角色:

- **Host**: 用戶接觸的 LLM 應用 (IDE、聊天應用、CLI harness)。
- **Client**:host 內部的協議端點, 與一個 server 維持 1:1 連接。
- **Server**: 暴露能力的一方, 可以是本地進程或遠端服務。

Server 向 client 提供三類原語:

- **Tools(工具)**: 模型控制的可執行能力——由模型決定何時呼叫。這是與第五章工具設計直接銜接的部分:MCP 工具的 name/description/inputSchema 完全適用第五章的全部 ACI 原則, 外加協議特性——工具註解 (read-only/destructive hints)、結構化輸

出 (`structuredContent` + `outputSchema`,2025-06-18)、資源鏈接 (工具結果引用資源而非內嵌大塊資料, 同版)。

- **Resources(資源)**: 應用控制的上下文——由 host 決定注入什麼 (檔案、文檔、資料庫 schema)。
- **Prompts(提示)**: 用戶控制的模板——由用戶顯式選用 (斜杠指令的協議化)。

三類原語的控制權劃分 (模型/應用/用戶) 是 MCP 架構中最被低估的設計: 它在協議層編碼了「誰有權決定什麼進入上下文」, 對應第六、七章的上下文治理職責分工。

Client 側原語 (sampling——server 反向請求 host 做推理;roots——host 向 server 聲明檔案範圍;elicitation——server 中途向用戶請求結構化輸入,2025-06-18 引入) 在 2026-07-28 版中命運分化:elicitation 存續並增強, 而 sampling 與 roots 被標記棄用 (見 9.3)。

傳輸層: 本地整合用 stdio(server 作為 host 的子進程); 遠端用 Streamable HTTP——單一 endpoint 接受 POST、可選 SSE 串流回應,2025-03-26 版取代了原始的 HTTP+SSE 雙端點設計。

9.2 修訂史: 一部濃縮的分散式系統課

MCP 的五個修訂版, 每一版都在糾正一類經典錯誤, 值得按時間軸精讀:

2024-11-05(初版): 核心架構落地。三原語、能力協商、list-changed 通知。

2025-03-26: 補上兩個生產化短板——**OAuth 2.1 授權框架** (遠端 server 的身份與授權) 與 **Streamable HTTP**(取代難以負載均衡的雙端點 SSE); 引入工具註解; 引入 JSON-RPC batching(注意此項的後續命運)。

2025-06-18: 安全大修。MCP server 被正式歸類為 **OAuth 2.0 Resource Server**, 以 RFC 9728(Protected Resource Metadata) 做授權服務器發現; **強制 client 使用 RFC 8707 Resource Indicators**——token 綁定目標 server, 杜絕「拿着 A 服務的 token 去存取 B」的 confused deputy 變體。功能面引入 elicitation、結構化工具輸出、資源鏈接。同時**刪除了上一版才引入的 batching**——一個罕見而健康的信號: 規格委員會願意承認並移除錯誤設計, 而非向後兼容地背負它。

2025-11-25: 規模化修訂,SEP(Specification Enhancement Proposal) 流程的第一個大版本。授權再演進(OIDC Discovery、Client ID Metadata Documents 取代對動態註冊的依賴、增量 scope 同意);**tasks(實驗性)**——長時任務的可輪詢句柄;sampling 獲得工具呼叫能力;工具命名指引、驗證錯誤返回給模型 (SEP-1303)、JSON Schema 2020-12 預設方言; 治理正式化 (工作組、SDK 分層)。

2026-07-28: **無狀態化轉向**。移除 initialize 握手與會話機制, 以每請求攜帶的 header(`Mcp-Method`、`MCP-Protocol-Version` 等) 取代——任何 server 實例可服務任何請求, 普通輪詢負載均衡即可水平擴展,sticky session 與共享會話存儲的要求被消滅。同時引入**擴展框架** (反向 DNS 命名、獨立版本化; 首批官方擴展:MCP Apps——server 渲染的沙箱化 HTML UI;Tasks 從實驗性畢業並無狀態化重構), 並**棄用 roots、sampling、logging**(≥12 個月移除窗口)。

這部修訂史的三條教訓具有超越 MCP 的普遍性:

1. **有狀態協議終將向無狀態演化**——會話親和性是

水平擴展的敵人,A2A v1.0 同期做了同樣的「web-aligned」轉向。

2. **授權的複雜度只增不減**——從無到 OAuth 2.1 到 RFC 9728/8707 到 CIMD, 每一步都由真實攻擊 (9.4 節) 驅動。
3. **協議應該敢於刪除**——batching 引入一版即刪;sampling 剛獲增強即被棄用。與 harness 的削減式哲學同構: 規格同樣「編碼了會過時的假設」。

治理:2025 年 12 月 9 日,Anthropic 將 MCP 捐贈給 Linux Foundation 旗下新成立的 Agentic AI Foundation(AAIF), 與 OpenAI 捐贈的 AGENTS.md、Block 捐贈的 goose 同為錨定專案; 白金會員含 AWS、Anthropic、Google、Microsoft、OpenAI 等。競爭廠商共同治理連接層, 表明工具連接已被行業共識視為非差異化的公共基礎設施。

9.3 官方 Registry 與生態

MCP Registry(registry.modelcontextprotocol.io,2025 年 9 月預覽) 是純元資料目錄:`server.json` 描述、反向 DNS

命名空間、GitHub/DNS 所有權驗證; 定位是供下游聚合器 (各廠 marketplace) 消費的上游源, 而非 host 直連的應用商店; 安全掃描責任明確下放給包註冊表與聚合器。工程含義:**registry 解決發現, 不解決信任**——你在任何 marketplace 安裝的 MCP server 依然是在給你的 harness 添加任意程式碼, 信任評估 (來源、簽名、權限面) 是使用方的責任, 這直接連到下一節。

9.4 安全: 協議層的威脅與規定

MCP 的 Security Best Practices 文件 (2025-06-18 設立,2025-11-25 大幅擴充) 是理解「代理連接外部服務」威脅面的最佳單一文獻。核心條目:

- **Confused deputy**:MCP 代理服務器以靜態 client ID 對接第三方授權服務器時, 攻擊者可借動態註冊與殘留的同意 cookie 跳過用戶同意、竊取授權碼。規定: 每 client 強制同意、redirect URI 精確匹配、`__Host-` 前綴簽名 cookie、嚴格 state 處理。
- **Token passthrough 禁令**:「MCP server 絕不可接受並非顯式簽發給它的 token」——必須做 audience

驗證。這條禁令的違反曾是早期生態的普遍實踐 (拿用戶的上游 API token 直接轉發), 它破壞審計鏈、繞過限流、混淆信任邊界。

- **會話劫持**:server 不得以 session 作身份認證;session ID 必須非確定性並綁定用戶 (`<user_id>:<session_id>` 模式); 共享事件隊列的注入變體亦在此節。
- **SSRF via 發現機制** (2025-11-25 新增): 惡意 server 把 OAuth 發現 URL 指向雲元資料端點 (169.254.169.254); 規定 HTTPS 強制、私網段封鎖、egress 代理。
- **工具投毒 (tool poisoning)**: 惡意指令藏於工具描述/元資料中, 隨定義進入模型上下文——2025 年 4 月由 Invariant Labs 命名, 現為 OWASP MCP Top 10 的 MCP03; 變體包括 rug-pull(安裝後改寫定義)與跨 server 遮蔽。防線: 安裝時凍結並審查工具定義、定義變更告警、host 對描述做注入掃描; 根本上, 工具描述必須被視為不可信輸入 (第十九章的信任邊界圖將涵蓋此路徑)。

對 harness 建造者的操作性總結:MCP server 的接入評審

應覆蓋四問——來源可信否 (供應鏈)、定義凍結否 (投毒)、token audience 正確否 (passthrough)、權限面最小否 (scope 與工具註解)。

9.5 工程實務:MCP 在 harness 中的正確位置

綜合第五、七、八章的機制,2026 年的成熟用法可歸納為:

1. **MCP 負責連接, 不負責全部介面**。經 MCP 接入的第三方工具往往是「API 鏡像」風格 (第五章批判的形態); 高頻工作流仍應在 harness 層以精心設計的一級工具封裝,MCP 工具作為長尾能力。
2. **規模治理交給 deferred loading**。多 server 環境的工具定義開銷以 tool search(第五章) 治理; 或走「工具即程式碼 API」路線 (第五章 5.6), 把 MCP server 呈現為檔案系統中的型別化模組, 以代碼執行聚合——兩者都已被量化驗證 (85% 與 98.7%/37% 的削減)。
3. **本地 stdio server 是特權程式碼**。它以 host 的權限運行, 安裝一個 stdio server 等於安裝一個本地應

用; 沙箱 (第八章) 與供應鏈審查同等適用。

4. **遠端 server 走完整 OAuth**,harness 在沙箱外完成授權流程,resource-scoped token 注入呼叫 (第八章憑證架構)。
5. **追蹤 spec 版本**。MCP 尚在快速演化 (棄用窗口 12 個月),harness 對 server 的版本協商與能力探測應防禦性編寫;2026-07-28 的無狀態語義簡化了自建 client 的難度, 新實現應直接以該版為基線。

9.6 本章小結

- MCP 以 host/client/server + tools/resources/prompts 把 M×N 整合問題降為 M+N; 三原語的模型/應用/用戶控制權劃分是協議層的上下文治理。
- 五個修訂版的主線: 授權持續硬化 (OAuth 2.1→RFC 9728/8707→CIMD)、傳輸走向無狀態 (Streamable HTTP→2026-07-28 徹底去會話)、敢於刪除 (batching、sampling/roots 棄用)。
- 治理中立化 (AAIF/Linux Foundation) 標誌連接層成為公共基礎設施;registry 解決發現而非信任。
- 安全規定的四大支柱:confused deputy 防護、to-

ken passthrough 禁令、會話反劫持、工具投毒防線; 接入評審四問 (來源、凍結、audience、最小權限)。

- 實務定位:MCP 管長尾連接, 一級工作流仍需 harness 層 ACI; 規模靠 deferred loading 或代碼執行聚合;stdio server 視同本地特權應用。

本章參考文獻

1. Model Context Protocol 規格與各版 changelog. https://modelcontextprotocol.io/specification/
2. MCP Security Best Practices(2025-11-25 版). https://modelcontextprotocol.io/specification/2025-11-25/basic/security_best_practices
3. MCP 2026-07-28 release candidate 公告. https://blog.modelcontextprotocol.io/posts/2026-07-28-release-candidate/
4. Linux Foundation, “Announcing the Agentic AI Foundation”, 2025-12-09.
5. MCP Registry. https://registry.modelcontextprotocol.io

6. Anthropic, “Code Execution with MCP”, 2025-11-04.
7. OWASP, MCP Top 10(MCP03: Tool Poisoning). https://owasp.org/www-project-mcp-top-10/
8. Simon Willison, “Model Context Protocol has prompt injection security problems”, 2025-04-09. # 第十章 Agent Skills: 程序性知識的封裝

MCP 解決「代理如何連接到能力」;Agent Skills 解決一個不同的問題:「代理如何知道某類工作該怎麼做」。Anthropic 於 2025 年 10 月 16 日發布 Skills, 同年 12 月 18 日將其發布為開放規格 (agentskills.io); 到 2026 年中, 它已被 Microsoft、OpenAI、Google、JetBrains、Cursor 等約三十餘個工具採納, 成為與 MCP、AGENTS.md 並列的第三個代理生態開放格式。本章解析其設計、與相鄰機制的精確分界, 以及編寫與治理的工程實務。

10.1 動機: 程序性知識的載體問題

模型缺的往往不是能力而是**做法**: 公司的品牌規範、法務審閱的步驟順序、某個內部系統的操作慣例、處理

PDF 表單的可靠流程。這類程序性知識 (procedural knowledge) 在 Skills 之前只有兩個去處, 而兩者都錯:

- **塞進系統提示/指令檔**: 永久佔用注意力預算, 且絕大多數會話用不到 (第三章的干擾項效應);
- **做成工具**: 工具的介面是函數簽名, 而「怎麼做一類事」是文檔加腳本加判斷, 強行函數化會丟失其結構。

Skills 的解法是**惰性載入的知識包**:「組織化的資料夾, 內含指令、腳本與資源, 代理可以動態發現並載入」。Anthropic 的比喻精確:「為代理編寫 skill, 就像為新同事編寫入職指南」。

10.2 規格解剖

一個 skill 是一個目錄, 核心是 `SKILL.md`:

```
---
name: log-sanitizer
description: Sanitize application logs by masking emails, tokens,
↪  API keys
  and secrets. Use before sharing logs externally or pasting into
↪  LLM context.
```

```
compatibility: Requires Python 3.11+. Executes local scripts.
metadata:
  author: platform-team
  version: "2.1"
allowed-tools: Bash(python:*) Read
---

# Log Sanitizer

## When to use
Before logs leave the service team's trust boundary...

## Steps
1. Run `python scripts/sanitize_logs.py --input <in> --output
↪  <out>`
2. On failure, inspect stderr and retry once.
3. Report: output path, lines processed, redaction count.

## Safety
Never echo raw secrets in the final response. When unsure, redact.
```

Frontmatter 欄位 (規格 2025-12-18 版):`name`(必填,≤64 字符, 小寫字母數字連字號, 須與目錄名一致)、`descrip-`

`tion`(必填,≤1024 字符,「做什麼 + 何時用」)、可選的 `license`、`compatibility`(環境要求,≤500 字符)、`metadata`(任意字串映射)、實驗性的 `allowed-tools`(預授權工具清單)。目錄內可附 `scripts/`、`references/`、`assets/`。

Progressive disclosure: 三級披露

Skills 的核心機制是與注意力預算的精確對齊, 分三級:

- **Level 1(元資料)**: 所有已安裝 skill 的 name+description 常駐系統提示, 每個約 100 token——僅足以支持「何時觸發」的判斷。
- **Level 2(指令)**:SKILL.md 正文, 建議 <5k token / <500 行, 在 skill 被判定相關時才讀入。
- **Level 3(資源)**:references、表單模板、腳本, 按需讀取; **腳本可以被執行而從不進入上下文**——一個 500 行的 PDF 表單解析腳本, 運行它花 0 個上下文 token, 而讓模型「徒手」完成等價工作要生成數千 token 且結果不定。

Anthropic 的類比:「像一本組織良好的手冊——先目錄, 再具體章節, 最後詳細附錄」。這是第七章 JIT 原則對知識

載體的應用: 常駐的只有索引, 內容永遠按需。

Level 3 的可執行腳本值得單獨強調, 因為它是 Skills 與純文檔的本質差異: **能寫成確定性程式碼的步驟就不應讓模型逐步執行**。腳本的工程要求與第五章工具輸出原則同源——非互動式 (絕不 `input()`)、有 `--help`、錯誤訊息指令化、輸出結構化。

10.3 與相鄰機制的精確分界

四個概念的職責邊界, 以一張表釘死:

機制	回答的問題	載入方式	控制權
工具 (含 MCP tools)	「我能對世界做什麼操作」	定義常駐或 deferred loading	模型決定呼叫
MCP	「能力如何標準化地接入」	連接層, 非內容層	協議

機制	回答的問題	載入方式	控制權
指令檔 (AGENTS.md/CLAUDE.md)	「此專案/組織的通用規範」	每會話無條件載入	應用
Skill	「某類任務的做法」	按需三級披露	觸發判定 (模型)+ 載入 (harness)

兩條最常被混淆的邊界:

- **Skill vs 指令檔**: 判準是適用頻率。每個會話都相關的內容 (建置指令、repo 規範) 進指令檔; 僅特定任務類型相關的內容 (「如何製作簡報」「如何審合同」) 進 skill——放進指令檔就是對其他所有會話的干擾項稅。
- **Skill vs MCP**: 官方定位是互補——「MCP 提供連接性,skills 提供程序性知識」; 典型組合是 skill 的步驟中呼叫 MCP 工具 (「用 jira_search 找到 ..., 然後按以下規則分類」)。一個經 MCP 連上 Jira 的代理

知道**能**做什麼, 一個帶 triage skill 的代理才知道你們團隊**怎樣**做 triage。

10.4 編寫方法論

規格之外,Anthropic 的工程文獻沉澱了一套編寫紀律:

1. **失敗驅動, 而非預防性**。從代表性任務上的實測失敗出發提煉 skill——與指令檔 (第六章 Hashimoto 原則) 同構:skill 是失敗的沉澱物, 不是想像的百科。
2. **name 與 description 是觸發器, 按提示工程對待**。Level 1 只有它們可見, 描述必須同時回答「做什麼」與「何時用」; 過寬導致誤觸發 (干擾), 過窄導致漏觸發 (沉睡)。
3. **中等範圍原則**。一個 skill 對應一類連貫任務;「公司全部流程」不是一個 skill,「修一種 bug」也不值得一個 skill。內容超過 5k token 就該拆分 Level 3 或拆成多個 skill。
4. **讓代理自己寫**。「請 Claude 把它成功的做法與常見錯誤捕捉為可重用的上下文與程式碼」——與第五章「代理重寫工具」同屬一族: 改進沉澱為可審查的確定

性 artifact。代碼執行環境下, 模型自發地「把自己的程式碼持久化為可重用函數, 積累一個更高階能力的工具箱」(第五章 5.6),Skills 正是這一行為的規範化容器。

10.5 治理與安全

Skills 是「指令 + 可執行程式碼」的複合體, 其威脅面等同於安裝軟件加注入提示的並集:

- **供應鏈**: 第三方 skill 市場的審計結果不容樂觀——安全廠商抽樣分析發現約三分之一的市場 skill 存在缺陷, 其中數十個含惡意載荷; 學術研究 (ToxicSkills) 測得代理 repo 中 13.4% 的 skill 至少有一個嚴重問題。官方立場明確: 僅安裝可信來源的 skill, 安裝前審計。
- **指令注入面**:SKILL.md 正文進入上下文, 即是注入載體; 惡意 skill 可指示代理外洩資料或濫用 `allowed-tools` 預授權。企業級對策:skill 倉庫集中管理、納入 code review、`allowed-tools` 按最小權限審批、執行層沙箱兜底 (第八章——這正是「確定性邊界」

價值的又一例: 即使 skill 被投毒,egress 允許清單仍在)。

- **版本與漂移**:skill 是行為配置, 應與指令檔、工具定義一同納入「代理配置整體版本化」(第十八章)——一次 skill 更新等同一次行為部署, 理應過 eval 關卡。

治理現狀的一個注腳: 與 MCP、AGENTS.md 不同,Agent Skills 截至本書寫作時仍由 Anthropic 於 agentskills.io 發布規格、未捐贈中立基金會——多方分析者指出這一治理缺口。採納者眾多而治理單邊, 是評估長期綁定風險時應知悉的事實。

10.6 本章小結

- Skills 是程序性知識的惰性載入容器, 填補「系統提示太貴、工具太窄」之間的空檔; 規格核心是 SKILL.md + 三級 progressive disclosure(~100 token 元資料 / <5k 正文 / 按需資源與可執行腳本)。
- 可執行腳本是本質優勢: 確定性步驟零上下文成本地執行, 模型只負責判斷與銜接。
- 分界: 每會話相關 → 指令檔; 特定任務類 →skill; 連

接性 →MCP; 操作原語 → 工具。

- 編寫紀律: 失敗驅動、description 即觸發器、中等範圍、讓代理沉澱自己的做法。
- 安全:skill= 軟件 + 提示的複合威脅面; 可信來源、集中審查、最小預授權、沙箱兜底; 注意其規格治理仍屬單邊。

本章參考文獻

1. Anthropic, “Equipping Agents for the Real World with Agent Skills”, 2025-10-16(2025-12-18 更新). https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills
2. Agent Skills 規格. https://agentskills.io/specification
3. Anthropic, “Introducing Agent Skills”, 2025-10-16. https://www.anthropic.com/news/skills
4. Anthropic, “Code Execution with MCP”(能力沉澱與 SKILL.md 的關聯), 2025-11-04.
5. The New Stack, “Agent Skills: Anthropic’s

Next Bid to Define AI Standards”;Snyk/ToxicSkills 安全分析 (轉引, 數字以原始報告為準). # 第十一章 驗證機制: 回饋迴圈、Hooks 與自主性的上限

本章論證全書最重要的一個命題: **代理系統的自主性上限由驗證能力決定, 而非由模型能力決定**。模型再強, 若系統無法廉價地判定其產出的對錯, 每一份產出都需要人工複核, 自主性就是幻覺; 反之, 驗證信號足夠強時, 較弱的模型也能經由迭代收斂到正確——這正是代理迴圈第三步「verify work」被 Anthropic 稱為「基本面上更可靠」之所在:「能夠檢查並改進自己輸出的代理, 從根本上更可靠」。

11.1 驗證不對稱性: 什麼任務適合代理

任務對代理的友好度, 取決於**生成與驗證的成本差**。判定一個 patch 是否通過測試套件是 O(秒) 的機器操作; 判定一份市場策略是否正確, 驗證成本不低於生成成本。前者存在強驗證信號, 代理可以自主迭代; 後者的迭代只是在無回饋地漫遊 (第四章反模式四)。

由此得出任務選型的驗證光譜:

- **強信號**: 編譯器、測試套件、型別檢查、schema 驗證、確定性的復現腳本——代理的黃金領域, 程式開發位於此。
- **中信號**: 視覺比對 (截圖 vs 設計稿)、性能指標、lint 與靜態分析——可自動化但有噪音。
- **弱信號**: 文風、策略合理性、「好不好看」——需要 LLM-as-judge 或人類, 誤差大。

harness 工程的一個高槓桿方向因此是**信號增強**: 為本來弱信號的任務建造強信號。Hashimoto 的表述:「你越是需要快速、高品質的工具來自動告訴它哪裏錯了」——他為自己的專案建造專用驗證工具, 正是把「代理寫得對不對」從人工判斷轉為機器判斷。OpenAI 團隊以自訂 linter 機械性強制分層架構, 同理: 架構正確性從評審意見變成 CI 信號。

11.2 三類回饋來源

Claude Agent SDK 的分類已成標準:

1. 規則式回饋 (rules-based):lint、測試、型別檢查、格式化器、自訂規則。特點: 確定、廉價、可無限重跑。工

程要點: 錯誤輸出要「帶解釋」——規則觸發時給出原因與修法方向, 回饋才可被模型有效消費 (與第五章錯誤訊息原則同源)。

2. 視覺回饋 (visual): 截圖、渲染結果、瀏覽器自動化。對前端與文檔類任務, 這是唯一忠實的信號——單元測試通過不等於頁面正確。長時程 harness 的實踐: 以 Puppeteer/Playwright MCP 驅動真實瀏覽器,「像用戶一樣測試」並截圖驗證; 已知盲區同樣被誠實記錄——瀏覽器原生 alert 模態框在該管道中不可見, 依賴它的功能 bug 率更高。工具層的對應演化:Playwright MCP 以**無障礙樹 (accessibility tree)** 而非像素作為預設表徵——結構化、確定、token 廉價, 視覺 (截圖) 模式按需啟用。

3. LLM-as-judge: 以另一個模型按 rubric 評分。適用於模糊標準, 但被明確標注為「三者中最不穩健」。使用紀律見 11.4。

11.3 確定性強制:Hooks 與關卡

指令是勸告, 模型可能忽略; 驗證要成為**強制**, 必須由確定性代碼執行。這是 2025 年 harness 設計的一個關鍵認知

轉變,Claude Code 的機制族是其代表實現:

- **Hooks**: 在生命週期事件 (PreToolUse、PostToolUse、Stop 等) 上運行的用戶定義程式, 可以攔截、改寫或否決代理行動。官方一語道破與指令檔的區別:「CLAUDE.md 是勸告性的,hooks 是確定性的」。典型用法:PreToolUse 攔截危險指令模式;PostToolUse 在每次編輯後自動跑格式化與 lint 並把失敗塞回上下文;Stop hook 在代理試圖結束會話時運行驗收腳本, 不通過則**拒絕結束** (連續 8 次受阻後才放行, 防死鎖)。
- **升級鏈**: 提示內建檢查 → `/goal`(獨立評估器逐輪覆核目標達成)→ Stop hook(確定性驗收)→ 對抗性驗證子代理。強度遞增, 按任務風險選檔。
- **要證據, 不要斷言**。最佳實踐文檔的措辭:「給 Claude 一個它能運行的檢查: 測試、build、可比對的截圖——這是『你盯着的會話』與『你可以走開的會話』之間的差別」。代理宣稱「測試已通過」不可接受,harness 應要求測試輸出本身作為證據。

同一思想的策略層版本是 **auto mode**(2026 年 3 月): 針對「用戶對 93% 的權限請求無腦批准」的 approval fatigue

現實, 以分類器取代人工逐條審批——服務端注入探測器檢查工具**輸出** (輸入側),Sonnet 4.6 transcript 分類器檢查**行動** (輸出側), 兩段式 (快速單 token 過濾 8.5% 誤報 → 僅對受標記行動做 CoT 覆核, 最終誤報 0.4%, 真實過激行為漏報 17%——官方稱之為「誠實的數字」)。設計上最值得學習的細節: 分類器只看用戶訊息與裸工具呼叫, **代理的自我辯解與工具輸出被剝離**——「評判者不能被說服」。這是驗證獨立性原則 (下節) 在權限層的實現。

11.4 自我評估問題與評判獨立性

模型評判自己的產出存在系統性偏置。Anthropic 在前端生成實驗中的直接觀察:「模型傾向於讚美自己的工作, 即使其客觀上平庸」; 對策同樣直接:「把幹活的代理與評判的代理分開, 被證明是一根強槓桿」。其 2026 年 3 月的三代理架構 (Planner / Generator / Evaluator) 即以 GAN 式的生成器/判別器分工落實此原則,Evaluator 以 Playwright 驅動真實應用、按設計品質/原創性/工藝/功能性打分, 每個設計迭代 5–15 輪回饋。

LLM-as-judge 的工程紀律 (綜合 Anthropic 評估文獻與

多代理研究系統經驗):

1. **單一連續評分 rubric**(0.0–1.0) 比多評委面板更一致;
2. 評判提示與被評產出**隔離生成過程**——評判者不應看到被評者的自我辯解 (auto mode 的剝離設計);
3. 評判者本身需要校準: 以人工標注樣本定期審計評判器的偏差;
4. **人類評估捕捉自動化漏掉的東西**——多代理研究系統的例子: 自動評估未發現代理偏好 SEO 內容農場, 人工抽查發現了。

11.5 防篡改: 驗證資產的完整性

代理有動機 (目標函數意義上的) 繞過驗證: 最快讓測試變綠的方法是刪掉測試。長時程實驗中被反覆觀察到的失敗模式包括「測試/規格刪除」與「過早宣告勝利」。防線分層:

- **介面層**:feature 清單採用 JSON 而非 Markdown——「JSON 檔案被證明比 Markdown 更能抵抗不當修改」(對模型而言, 改動結構化資料的心理門檻高於

改散文; 這是 poka-yoke 的又一實例); 每個 feature 帶 `passes: false` 初始欄位與逐步驗證指引。

- **指令層**:「刪除或修改測試是不可接受的」寫入提示——必要但不充分。
- **確定層**: 驗證資產 (測試目錄、CI 配置、feature 清單) 以 hooks 或檔案權限設為代理不可寫;CI 在代理環境之外運行, 結果不可偽造。
- **審計層**:Carlini 的 C 編譯器專案的教訓:「任務驗證器必須近乎完美, 否則 Claude 會解決錯誤的問題」——16 個並行代理兩週無人值守, 前提是 GCC 作為「已知正確的參照編譯器」提供了不可欺騙的 oracle, 且測試以確定性子採樣 (1%/10%) 控制成本。驗證器的品質上限就是無人值守的時長上限。

11.6 評估器經濟學: 何時值得建

驗證組件自身有成本 (Evaluator 代理的推理費、CI 時長、維護)。Anthropic 的量化例子給出了決策框架:Retro Game Maker 任務, 單代理 20 分鐘 $9; 全 harness(含 Evaluator)6 小時 $200——換來的是顯著更高的品質與長時程的無人值守能力。結論不是「總是值得」, 而是: **評估**

器只在其把關的能力超出「當前模型可靠獨立完成」的邊界時才回本——又一次, 削減式哲學: 模型每前進一代, 某些驗證組件就從必需品降格為開銷, 應被重新評估。可操作的判準:

- 任務短、失敗廉價、人在迴路 → 最小驗證 (規則式即可);
- 任務長、無人值守、失敗昂貴 → 全鏈驗證 (規則 + 視覺 + 獨立評判 + 防篡改);
- 介於其間 → 以 pass^k 實測 (第十七章) 決定驗證投資的邊際收益。

11.7 本章小結

- 自主性上限 = 驗證能力; 任務選型看生成/驗證成本差; 高槓桿方向是為弱信號任務建造強信號 (專用驗證工具、機械化架構規則)。
- 三類回饋: 規則式 (確定、廉價, 錯誤要帶解釋)、視覺 (前端唯一忠實信號, 注意管道盲區)、LLM-as-judge(最弱, 需紀律)。
- 勸告與強制分離: 指令檔勸告,hooks 確定性強制; 升

級鏈按風險選檔; 要證據不要斷言;auto mode 展示了「不可被說服的評判者」的設計。

- 自我評估偏置是實測現象; 生成與評判必須分離; 評判者要校準, 人類抽查兜底。
- 防篡改分四層 (介面/指令/確定/審計); 驗證器品質上限 = 無人值守時長上限。
- 評估器按經濟學決策: 只在把關能力超出模型可靠邊界時回本, 隨模型演進重新評估。

本章參考文獻

1. Anthropic, “Building Agents with the Claude Agent SDK”(三類回饋), 2025-09-29.
2. Claude Code Best Practices(hooks、升級鏈、證據原則). https://code.claude.com/docs/en/best-practices
3. Anthropic, “How We Built Claude Code Auto Mode”, 2026-03-25. https://www.anthropic.com/engineering/claude-code-auto-mode
4. Anthropic, “Harness Design for Long-Running Application Development”(自我評估問題、Eval-

uator 經濟學), 2026-03-24.

5. Anthropic, “Effective Harnesses for Long-Running Agents”(JSON feature 清單、失敗模式表), 2025-11-26.
6. Nicholas Carlini / Anthropic, “Building a C Compiler with a Team of Parallel Claudes”(驗證器完美性), 2026-02-05.
7. Mitchell Hashimoto, “My AI Adoption Journey”(驗證工具建造), 2026-02-05.
8. Anthropic, “How We Built Our Multi-Agent Research System”(LLM-as-judge 紀律), 2025-06-13. # 第三部架構模式 {.part}

第十二章工作流模式的實現:Chaining、Routing、Parallelization 與其組合

第四章建立了五種 workflow 模式的概念框架; 本章給出它們的參考實現與工程細節。示例採用廠商中立的 Python: 假定存在一個 `llm(messages, *, schema=None, model=None)` 函數封裝任意供應商的 API(schema 參數啟用 structured output), 不依賴任何編排框架——目的正是展示這些模式的本質只是普通程式, 框架提供的是持久化與工程便利 (第十五章), 而非魔法。

```
from dataclasses import dataclass
from concurrent.futures import ThreadPoolExecutor
import json
```

```
def llm(messages: list[dict], *, schema: dict | None = None,
        model: str = "default") -> str | dict:
    """ 呼叫任意 LLM API;schema 非空時以 structured output 強制返
    ↪ 回合法 JSON。"""
    ...
```

12.1 Prompt Chaining: 分解與關卡

結構: 固定順序的步驟, 每步之間插入程式化關卡 (gate)。

示例: 技術文檔的「生成 → 事實核驗 → 潤色」鏈。

```
@dataclass
class ChainResult:
    ok: bool
    output: str
    failed_gate: str | None = None


def doc_chain(spec: str, facts: list[str]) -> ChainResult:
    # 步驟 1: 生成初稿
    draft = llm([{"role": "user", "content":
        f" 依據以下規格撰寫技術文檔草稿:\n{spec}"}])


    # 關卡 1(程式化): 結構完整性——確定性檢查, 不花推理
```

```
    required_sections = ["## 概述", "## 配置", "## 故障排除"]
    if not all(s in draft for s in required_sections):
        return ChainResult(False, draft, "structure_gate")

    # 步驟 2: 事實核驗 (模型), 輸出 schema 化
    verdict = llm(
        [{"role": "user", "content":
          f" 逐條核對文檔是否與事實清單一致。\n事實:{facts}\n文
          ↪  檔:{draft}"}],
        schema={"type": "object", "properties": {
            "consistent": {"type": "boolean"},
            "violations": {"type": "array", "items": {"type":
            ↪  "string"}}},
            "required": ["consistent", "violations"]})

    # 關卡 2: 核驗結果驅動的分支
    if not verdict["consistent"]:
        draft = llm([{"role": "user", "content":
            f" 修正以下違例後重新輸出全
            ↪  文:{verdict['violations']}\n\n{draft}"}])

    # 步驟 3: 潤色
    final = llm([{"role": "user", "content": f" 潤色以下文檔, 保
↪   持技術內容不變:\n{draft}"}])
```

```
    return ChainResult(True, final)
```

工程要點:

1. **關卡優先確定性**。能用字串匹配、schema 驗證、規則引擎判定的, 絕不花一次推理。模型關卡 (事實核驗) 的輸出必須 structured——`consistent` 是布爾值而非讓下游解析自由文字。
2. **鏈長即方差**。每一步都在傳遞上游的殘餘誤差; 超過四五步的鏈應審視是否有步驟可合併, 或是否其實需要 agent(運行時決定步數)。
3. **中間產物落盤**。每步輸出寫入檔案 (或框架的 checkpoint), 失敗時從斷點重試而非全鏈重跑——這是第十五章持久化的最小手工版。

12.2 Routing: 分類即架構

結構: 一次廉價的分類推理決定後續路徑。示例:IT 服務台的工單分診。

```
ROUTES = {
    "password_reset":  {"model": "small",   "handler":
    ↪  "runbook_flow"},
    "vpn_issue":       {"model": "small",   "handler":
    ↪  "runbook_flow"},
    "app_access":      {"model": "default", "handler":
    ↪  "approval_flow"},
    "incident":        {"model": "frontier","handler":
    ↪  "agent_flow"},   # 唯一的 agent 分支
    "other":           {"model": "default", "handler":
    ↪  "human_queue"},
}

def route(ticket: str) -> str:
    r = llm(
        [{"role": "user", "content": f" 將工單分類。\n工
↪   單:{ticket}"}],
        schema={"type": "object", "properties": {
            "category": {"enum": list(ROUTES.keys())},
            "confidence": {"type": "number"}},
            "required": ["category", "confidence"]},
        model="small")
    # 低置信度回退，而非硬走錯誤分支
    return "other" if r["confidence"] < 0.7 else r["category"]
```

工程要點:

1. **枚舉封閉**。`category` 是 enum,constrained decoding 保證不會出現第六個類別; 新增類別是 schema 變更, 走版本化流程。
2. **置信度回退**。分類器不確定時落入人工隊列, 優於自信地走錯——路由錯誤的代價是整條下游路徑的浪費。
3. **路由即成本策略**。密碼重置類走小模型跑 runbook, 事故類才動用前沿模型與 agent 分支。第二十章將量化: 合理分診常能削減一個數量級的推理成本。
4. **路由器可離線評估**。積累帶標注的分類數據集, 路由準確率是一個普通的分類指標——這是整個系統中最容易 eval 的組件, 應第一個建立迴歸測試 (第十七章)。

12.3 Parallelization:Sectioning 與 Voting

Sectioning——獨立子任務並行。示例:PR 的三維度並行評審。

```
REVIEW_DIMENSIONS = {
    "security":     " 審查以下 diff 的安全問題 (注入、越權、秘密洩露)…",
    "performance":  " 審查以下 diff 的性能問題 (N+1、鎖競爭、記憶體)…",
    "maintainability": " 審查以下 diff 的可維護性 (命名、耦合、測試覆蓋)…",
}
FINDING_SCHEMA = {"type": "object", "properties": {
    "findings": {"type": "array", "items": {"type": "object",
     "properties": {
        "file": {"type": "string"}, "line": {"type": "integer"},
        "severity": {"enum": ["high", "medium", "low"]},
        "claim": {"type": "string"}},
        "required": ["file", "claim", "severity"]}}},
    "required": ["findings"]}


def parallel_review(diff: str) -> list[dict]:
    with ThreadPoolExecutor() as pool:
        futures = {dim: pool.submit(
            llm, [{"role": "user", "content":
 f"{prompt}\n\n{diff}"}],
            schema=FINDING_SCHEMA)
            for dim, prompt in REVIEW_DIMENSIONS.items()}
```

```
        findings = []
        for dim, f in futures.items():
            findings += [{**x, "dimension": dim} for x in
↪   f.result()["findings"]]
    return dedupe_by_location(findings)   # 聚合層: 按 file+line
    ↪   去重
```

Voting——同一任務多次採樣後聚合, 換取信度。典型用法: 對 sectioning 產出的每條 finding, 派 N 個獨立「反駁者」驗證, 多數存活才上報:

```
def adversarial_verify(finding: dict, diff: str, n: int = 3) ->
↪   bool:
    prompt = (f" 嘗試反駁以下代碼評審發現。若無法確證其為真實問
↪   題, 判定 refuted。\n"
              f" 發現:{finding['claim']}\n代碼:{diff}")
    schema = {"type": "object",
              "properties": {"refuted": {"type": "boolean"},
                             "reason": {"type": "string"}},
              "required": ["refuted", "reason"]}
    with ThreadPoolExecutor() as pool:
        votes = [pool.submit(llm, [{"role": "user", "content":
↪   prompt}],
                             schema=schema) for _ in range(n)]
```

```
        survive = sum(not v.result()["refuted"] for v in votes)
    return survive >= (n // 2 + 1)
```

工程要點:

1. **反駁式提示**。驗證者的任務被框定為「嘗試反駁」而非「請確認」——對抗式框架顯著降低 LLM-as-judge 的迎合偏置 (第十一章)。「不確定時判 refuted」的預設進一步壓低假陽性, 適合評審這類假陽性昂貴 (浪費人類注意力) 的場景。
2. **視角多樣化優於同質冗餘**。當 finding 可能以多種方式為假時, 給 N 個驗證者不同的視角 (正確性/可復現性/影響面) 勝過三個相同提示——多樣性捕捉冗餘捕捉不到的失效模式。
3. **並行的邊界是獨立性**。三個評審維度互不依賴, 可並行;「先修 A 再修 B」的兩個修改不獨立, 強行並行會產生衝突輸出——這是第十三章多代理寫衝突的單機預演。

12.4 Orchestrator-Workers: 動態分解

結構: 協調者在運行時分解任務、派發、綜合。與 12.3 的區別: 子任務清單由輸入決定。

```
def orchestrate(task: str, worker_tools: list) -> str:
    plan = llm(
        [{"role": "user", "content":
          f" 將任務分解為可獨立執行的子任務，標注每個的目標、輸出格式與邊界。\n任務:{task}"}],
        schema={"type": "object", "properties": {
            "subtasks": {"type": "array", "maxItems": 8, "items": {
                "type": "object", "properties": {
                    "objective": {"type": "string"},
                    "output_format": {"type": "string"},
                    "boundaries": {"type": "string"}},
                "required": ["objective", "output_format", "boundaries"]}}},
            "required": ["subtasks"]})

    with ThreadPoolExecutor() as pool:
```

```
        results = list(pool.map(
            lambda st: run_worker(st, worker_tools),
            ↪ plan["subtasks"]))

    return llm([{"role": "user", "content":
        f" 綜合以下子任務結果，回答原任務。\n任務:{task}\n結
        ↪ 果:{json.dumps(results, ensure_ascii=False)}"}])
```

此模式的工程要點幾乎全部來自 Anthropic 多代理研究系統的實測教訓, 將在第十三章完整展開; 此處先記三條直接影響程式碼的:

1. **委派說明四要素**:objective、output_format、tool guidance、boundaries——缺任一項,workers 就會重複勞動或漏做。上面的 schema 把它結構化為強制欄位。
2. **努力度標定**: 協調者提示中應包含顯式的規模啟發式(「簡單事實查證:1 個 worker、3–10 次工具呼叫; 對比分析:2–4 個 worker」), 否則模型對任務規模的判斷方差極大。
3. **maxItems 上限**: 分解數量必須有 harness 強制的硬上限——否則某些輸入會誘發數十個子任務的爆炸

(成本失控的第一大來源, 見第二十章)。

12.5 Evaluator-Optimizer: 生成—評判迴圈

```
def refine(task: str, max_rounds: int = 4, threshold: float =
↪  0.85) -> str:
    draft = llm([{"role": "user", "content": task}])
    for _ in range(max_rounds):
        review = llm(
            [{"role": "user", "content":
              f" 按 rubric 評估並給出具體修改建議。\n"
              f"Rubric: 準確性 0.4 / 完整性 0.3 / 清晰度 0.3, 各
              ↪  項 0–1。\n"
              f" 任務:{task}\n產出:{draft}"}],
            schema={"type": "object", "properties": {
                "score": {"type": "number"},
                "actionable_feedback": {"type": "array", "items":
                ↪  {"type": "string"}}},
                "required": ["score", "actionable_feedback"]},
            model="frontier")          # 評判者用更強的模型
        # 終止條件:達標或回饋枯竭
        if review["score"] >= threshold or not
        ↪  review["actionable_feedback"]:
```

```
            break
        draft = llm([{"role": "user", "content":
            f" 依據回饋修改。\n回
            ↪  饋:{review['actionable_feedback']}\n原
            ↪  文:{draft}"}])
    return draft
```

工程要點:

1. **rubric 顯式加權**, 單一連續分數 (第十一章: 單評分比多評委一致)。
2. **評判者 ≠ 生成者**: 不同的提示上下文, 理想情況下不同的模型; 評判提示中不包含生成過程的自我辯解。
3. **雙重終止條件**: 輪數上限 (成本兜底)+ 分數/回饋枯竭 (收斂判定)。缺前者是無限迴圈事故的經典成因; 缺後者是在浪費已收斂後的輪次。
4. **監控分數軌跡**: 若多輪分數不升, 說明評判標準模糊或任務超出模型能力——此時增加輪數無益, 應改進 rubric 或升級模型。

12.6 組合: 一個完整的評審管線

五種模式的真實價值在組合。以下是一條生產形態的 PR 評審管線, 標注每段所用模式:

```
PR diff
  |
  ├─[Routing] 按 diff 規模與涉及路徑分級:
  |     docs-only → 快速通道(小模型單步)
  |     normal    → 標準管線
  |     high-risk(觸及 auth/、payments/)→ 加強管線
  |
 ├─[Parallelization/sectioning] 安全、性能、可維護性三維並行
  |       (各維度可用子代理隔離上下文——第十三章)
  |
  ├─[聚合 + 確定性去重](純程式碼,無推理)
  |
 ├─[Parallelization/voting] 每條 finding 三票對抗驗證,多數存活
  |
 ├─[Prompt chaining] 存活 finding → 生成修復建議 → schema 驗證 -
  |
```

```
└─[規則式關卡] SAST/secret scanning 的確定性結果直接合併
    (deterministic scanner 與 AI reviewer 並置:語義層歸模型,
        不可妥協規則歸掃描器)
```

這條管線體現的分工原則值得成為定式:**AI 評審員負責語義層、設計層、可讀性層; 確定性掃描器負責不可妥協的規則檢查**。兩者不是競爭關係——掃描器的每一條規則, 都是從模型評審職責中移除的一項方差來源。

12.7 本章小結

- 五種模式的實現核心:chaining 靠確定性關卡與落盤斷點;routing 靠封閉枚舉、置信度回退與成本分診;sectioning 靠獨立性判定,voting 靠對抗式提示與視角多樣化;orchestrator 靠四要素委派、努力度標定與硬上限;evaluator-optimizer 靠顯式 rubric、評判分離與雙重終止。
- 模式皆是普通程式:structured output + 線程池即可完整表達; 框架的價值在持久化與運維, 不在模式本身。
- 組合管線的分工定式: 語義判斷歸模型 (且經對抗驗

證), 可判定規則歸確定性工具。

本章參考文獻

1. Anthropic, “Building Effective Agents”(五模式定義), 2024-12-19.
2. Anthropic, “How We Built Our Multi-Agent Research System”(委派四要素、努力度標定), 2025-06-13.
3. LangGraph 文檔 (workflow 模式參考實現與 Send API). https://docs.langchain.com/
4. SonarSource, “Quality gates for AI-generated code”(確定性掃描器與 AI 評審並置的實務依據). # 第十三章多代理系統: 隔離、共享與協調的代價

多代理是這個領域被炒作與被誤解得最嚴重的架構。2025 年, 兩篇立場相反的重量級文獻——Anthropic 的多代理研究系統報告與 Cognition 的〈Don’t Build Multi-Agents〉——同時成為必讀; 同年, 首個多代理失敗實證研究 (MAST) 給出了系統性的失敗分類。到 2026 年, 塵埃初定: 多代理有效的條件已經可以被精確陳述。本章的任務就是陳述它。

13.1 收益的實證:Anthropic 研究系統

Anthropic 為 Claude 的 Research 功能建造的 orchestrator-workers 系統, 是多代理收益的最強公開實證:

- 架構:lead agent(Opus 4) 規劃並派生並行 sub-agents(Sonnet 4) 搜索, 另有獨立的 citation agent 做引文歸屬。
- 效果: 在內部研究評估上, 多代理配置比單代理 Opus 4 **高出 90.2%**; 並行工具呼叫把複雜查詢的研究時間縮短至多 90%。
- 歸因分析:BrowseComp 型評估上,**token 用量解釋了 80% 的性能方差**——多代理勝出的機制不神秘, 就是有效 token 吞吐的擴張: 多個乾淨窗口並行閱讀, 遠超單窗口的注意力預算所能承載。

同一報告誠實給出了代價: 代理消耗約為對話的 4 倍 token, **多代理系統約為對話的 15 倍**。結論由此而來: 多代理只適用於「價值足以支付 15 倍成本、且可並行化」的任務。

13.2 反方立場:Cognition 的共享上下文原則

Cognition(Devin 的開發商) 的 Walden Yan 在 2025 年 6 月給出了對立面, 其論證圍繞一個被多代理熱潮忽視的事實: **行動攜帶隱含決策**。

兩個並行 subagent 各自實現遊戲的一部分, 各自在過程中做了數十個未被記錄的微觀決策 (命名、資料結構、風格); 聚合時, 這些決策互相衝突——他們的 Flappy Bird 例子中, 兩個 subagent 交回的是視覺與架構上不相容的兩半。由此得出兩條原則:

1. **共享完整的代理軌跡, 而非摘要**——「share full agent traces, not just individual messages」。摘要必然丟失隱含決策。
2. **預設單線程線性代理**; 超長任務用專門的壓縮模型蒸餾歷史, 而不是切分給多個代理。

值得記錄的後續:2026 年初,Yan 公開軟化立場——「一年前我會勸人不要建多代理……今天我們找到了一些確實有效的配置」。這不是立場反轉, 而是條件的澄清: 他反對的

配置 (並行寫入、摘要傳遞決策) 仍然錯誤; 有效的配置符合下文的收斂結論。

13.3 失敗的分類學:MAST

Cemri 等人的〈Why Do Multi-Agent LLM Systems Fail?〉(NeurIPS 2025) 對 7 個流行多代理框架的 200 餘條執行軌跡做了標注分析, 得出 14 種失敗模式、三大類:

1. **規格與系統設計失敗**: 角色定義含糊、任務規格缺失約束、架構與任務不匹配;
2. **代理間錯位 (inter-agent misalignment)**: 資訊不同步、假設分歧、重複勞動、互相等待;
3. **任務驗證與終止失敗**: 無人負責驗收、過早終止、無限循環。

其核心論斷與本書的框架完全一致: **多數失敗是組織性/harness 性失敗, 而非模型能力失敗**——壞的角色規格、缺失的驗證, 恰是第四、十一章討論的 harness 缺陷在多代理拓撲上的放大。多代理不會修復單代理 harness 的缺陷, 只會給它乘上代理數。

13.4 收斂: 什麼時候多代理是對的

把三方證據疊加,2026 年的共識 (Anthropic 在〈When to use multi-agent systems〉中的正式表述) 可以壓縮為三個有效動機與一條分解準則:

三個有效動機——多代理僅在以下情形有淨收益:

1. **上下文保護 (context protection)**: 偵察與探索的大量讀取污染主窗口, 隔離到子代理、摘要回傳 (第七章 7.4);
2. **可並行的探索 (parallelizable exploration)**: 研究、搜索、評審這類**讀密集**且各分支獨立的任務;
3. **特化 (specialization)**: 不同子任務需要截然不同的工具集/提示/模型, 合併在單代理中互為干擾項。

一條分解準則——「按上下文需求分解, 而非按問題類型分解」: 同一個 feature 的設計、實現、測試共享大量上下文, 拆給三個「角色」代理只會製造 Cognition 式的決策失散; 而十個互不相關的檔案遷移各自獨立, 天然可並行。擬人化的組織圖 (PM/架構師/工程師開會) 是按問題類型分解的典型錯誤 (第四章反模式三)。

讀寫定律——綜合雙方立場的最實用口訣: **並行化讀, 序列化寫**。探索、檢索、評審 (讀) 可以也應該並行; 修改、決策、產出 (寫) 必須經過單一上下文串行化。Anthropic 與 Cognition 在此完全一致: 前者的 subagents 只做搜索、綜合由 lead 完成; 後者反對的正是並行寫。最小可靠的多代理模式因此是: **主代理 + 驗證子代理**——寫者唯一, 第二個代理只做獨立評判 (第十一章的評判獨立性)。

13.5 委派工程: 讓 orchestrator 稱職

多代理的品質瓶頸幾乎總在協調層。可操作的委派紀律 (源自 Anthropic 的實測迭代):

1. **四要素委派**: 每個 subagent 收到 objective、output format、工具指引、邊界 (boundaries)。缺 boundaries 時, 兩個 subagent 會各自「順便」做了同一件事。
2. **努力度標定**:orchestrator 的提示需包含規模啟發式 (簡單查證 1 agent/3–10 calls; 對比 2–4 agents/各 10–15 calls), 否則對簡單問題也會派出艦隊。
3. **回傳格式合同**:subagent 的最終訊息就是返回值,

應為結構化資料而非對話文; 典型摘要預算 1,000–2,000 token(第七章)。

4. **廣度優先再收窄**: 搜索型 subagent 先廣後窄的策略應寫入其提示——模型的自然傾向是過早深潛。
5. **失敗也要回傳**:subagent 的錯誤與空手而歸必須顯式回報,orchestrator 依此重派; 靜默失敗是 MAST 第二類失敗 (資訊不同步) 的主因之一。

13.6 通訊拓撲與 A2A 協議

組織內的多代理通訊, 以上述 orchestrator 星形拓撲為主流; 跨組織、跨廠商的代理互操作則在協議化。**A2A(Agent2Agent)**:Google 2025 年 4 月發起, 同年 6 月捐贈 Linux Foundation,2026 年 4 月達到 v1.0 穩定版、150+ 支持組織, 已有供應鏈、金融、保險領域的生產部署。

架構要點:**Agent Card**(位於 `/.well-known/agent-card.json` 的 JSON 自述: 能力、技能、端點、認證要求;v1.0 增加密碼學簽名以跨組織驗明身份)、**任務生命週期** (submitted → working → input-required

→ completed/failed/canceled, 交換 Message 與 Artifact, 支持 SSE 串流與推送通知)、多傳輸 (JSON-RPC/gRPC/REST)。

與 MCP 的分工一句話:**MCP 縱向連工具,A2A 橫向連代理**——A2A 的對端是不透明的對等代理 (不共享內部記憶、工具與邏輯), 各對端內部可自行使用 MCP。設計上的相似演化也值得注意:A2A v1.0 與 MCP 2026-07-28 同步走向「web-aligned」的無狀態、可負載均衡形態 (第九章的教訓一)。

對多數團隊的務實建議: 組織內部用 harness 原生的 subagent 原語 (進程內、共享 trace、低開銷);A2A 保留給真正的組織邊界——對方的代理不是你部署的、不能共享你的信任域時, 才需要協議級的互操作。

13.7 案例:16 個並行 Claude 建造 C 編譯器

Carlini 的實驗 (2026 年 2 月) 是「讀寫定律」與最小協調的極限測試:16 個 Opus 4.6 代理, 各在獨立 Docker 容

器, 兩週無人值守, 產出 10 萬行 Rust 的 C 編譯器——能在 x86/ARM/RISC-V 上編譯 Linux 6.9,GCC torture suite 通過率 99%, 總成本約 $20,000 API 費、約 2,000 個會話、20 億輸入 token。

其協調機制的簡陋是刻意的: **以 git 為鎖服務**——代理透過向 `current_tasks/` 目錄寫檔案認領任務,git 同步衝突即仲裁; 無中央 orchestrator, 無訊息隊列。使之可行的不是協調的精巧, 而是任務結構與驗證的品質:

- 任務天然可分區 (不同的編譯器測試群), 寫衝突面小;
- **oracle 近乎完美**:GCC 作為參照編譯器提供不可欺騙的差分測試信號 (第十一章: 驗證器品質上限 = 無人值守時長上限);
- 測試輸出保持簡潔、細節落盤 (第七章卸載); 確定性子採樣 (1%/10%) 控制驗證成本;
- 湧現的特化: 部分代理自發長期扮演去重、性能、批評、文檔角色——特化可以湧現, 無需預先指派人格。

這個案例的正確讀法不是「多代理很強」, 而是: **當任務可分區、驗證近乎完美、寫入經 git 串行化時, 協調可以退化**

到檔案系統級的原語。三個前提缺一, 同樣的架構就會退化為 MAST 的失敗目錄。

13.8 本章小結

- 收益機制: 多窗口並行擴張有效注意力預算 (token 用量解釋 80% 方差; 研究評估 +90.2%); 代價: 約 15 倍 token 與新增的協調失敗面 (MAST 三類十四種)。
- 有效動機僅三: 上下文保護、並行探索、特化; 分解按上下文需求而非問題類型; 擬人組織圖是反模式。
- 讀寫定律: 並行化讀、序列化寫; 最小可靠模式 = 主代理 + 驗證子代理。
- 委派工程五條: 四要素、努力度標定、回傳合同 (1–2k token 結構化)、先廣後窄、失敗顯式回報。
- 拓撲: 組織內用原生 subagent; 組織邊界用 A2A(Agent Card、任務生命週期、v1.0 簽名)。MCP 連工具,A2A 連代理。
- C 編譯器案例: 可分區任務 + 完美 oracle+git 串行化寫入, 協調可簡至檔案原語; 前提缺一即失敗。

本章參考文獻

1. Anthropic, “How We Built Our Multi-Agent Research System”, 2025-06-13.
2. Walden Yan / Cognition, “Don’t Build Multi-Agents”, 2025-06-12. https://cognition.com/blog/dont-build-multi-agents
3. Cemri et al., “Why Do Multi-Agent LLM Systems Fail?”(MAST), arXiv:2503.13657, NeurIPS 2025.
4. Anthropic / Claude Blog, “When to Use Multi-Agent Systems (and When Not To)”, 2026-01-23.
5. A2A Protocol(v1.0). https://a2a-protocol.org;Linux Foundation A2A 一週年公告, 2026-04-09.
6. Nicholas Carlini / Anthropic, “Building a C Compiler with a Team of Parallel Claudes”, 2026-02-05. # 第十四章長時程代理: 跨上下文窗口的工程

當任務的規模超過一個上下文窗口所能承載——以小時、

天甚至週計的自主開發——harness 面對的是一個質變的問題: 代理「必須以離散會話工作, 而每個新會話開始時對之前發生的一切毫無記憶」。長時程工程的全部技術, 都是在回答同一個問題: **如何讓一連串失憶的會話, 表現得像一個有記憶的工程師**。本章按時間順序追蹤這個問題在 2025–2026 年的三代解法, 結尾提煉出獨立於任何一代的持久原則。

14.1 失敗模式的基線

先確立無干預時會發生什麼。Anthropic 在多上下文窗口專案上的觀察 (2025 年 11 月), 即使是當時最強的 Opus 4.5 也呈現兩類對稱的失敗:

- **過度雄心 (over-ambition)**: 單次會話攬下過多工作, 在實現中途耗盡上下文, 留下一堆半成品 feature 與不可運行的狀態, 下一個會話接手的是廢墟;
- **過早收官 (premature completion)**: 淺嘗即宣告任務完成——第十一章的驗證缺失問題在長時程下的表現。

輔助性的失敗還有: 測試/規格刪除 (繞過驗證的捷徑)、環

境劣化 (壞狀態被後續會話繼承並疊加)、每個新會話重複摸索環境設置而空耗 token。這五項構成長時程 harness 的需求清單。

14.2 第一代:Initializer ／ Coder 雙角色 (2025-11)

〈Effective Harnesses for Long-Running Agents〉的設計哲學被明確表述為「複製高效軟件工程師每天在做的事」, 落為兩個角色:

Initializer agent(僅首個會話): 搭建環境——`init.sh`(啟動開發伺服器並做基本端到端驗證)、`claude-progress.txt` 進度日誌、初始 git commit, 以及 **JSON 格式的 feature 需求清單**:claude.ai 克隆實驗中超過 200 個 feature, 每個帶類別、描述、逐步驗證指引與初始為 `false` 的 `passes` 欄位。JSON 而非 Markdown 是防篡改選型 (第十一章)。

Coding agent(其後所有會話), 每會話遵循固定儀式:

啟動:讀 `claude-progress.txt` + `git log` → 運行 `init.sh` 驗證環境健康
工作:從清單挑選【恰好一個】未通過的 `feature` → 實現

驗證:端到端測試(Puppeteer MCP 驅動真實瀏覽器,截圖為證)

收尾:commit → 更新進度檔與 feature 清單 → 留下乾淨可合併的狀態

「每會話一個 feature」的粒度限制直接針對 over-ambition; 端到端瀏覽器驗證針對 premature completion;git + 進度檔構成跨會話記憶 (第六章結構化筆記的工程化);init.sh 消滅重複的環境摸索。五個失敗模式, 五個對應機制——這是失敗驅動設計 (第六章 Hashimoto 原則) 在系統層的教科書示範。

14.3 第二代:Planner ／ Generator ／ Evaluator(2026-03)

四個月後的〈Harness Design for Long-Running Application Development〉展示了下一代架構, 兩個變化最重要。

架構變化: 三代理分工——**Planner** 把一行需求擴展為完整產品規格;**Generator** 實現 (React/Vite/FastAPI 棧,git 管理);**Evaluator** 以 Playwright MCP 驅動真實應用、按設計品質/原創性/工藝/功能性評分, 每個設計迭代 5–15 輪

回饋。生成與評判的分離 (第十一章自我評估問題) 從原則升級為一級架構組件——一個 GAN 式的生成器/判別器結構。

削減變化:Opus 4.6 緩解了 context anxiety(第三章), 於是第一代中「為 4.5 的行為癖性建造」的組件被**刪除**——sprint 合約移除, 逐 feature 驗收改回單次端到端評估; 文中並精確區分了 context reset(結構化交接後全清) 與 compaction(原地摘要) 的不同適用場景 (第七章)。量化的代價收益:Retro Game Maker, 單代理 20 分鐘 $9, 全 harness 6 小時 $200;4.6 harness 建造一個 DAW 應用 3 小時 50 分、$124.70。結論句已在第一章引用:harness 的每個組件都編碼了對模型缺陷的假設,「這些假設值得被壓力測試」。

兩代之間的差分本身就是本書削減式論題的最佳資料點:**第一代的許多機制不是「長時程工程的永恆真理」, 而是「Opus 4.5 時代的補償」**。第 14.6 節將把兩者剝離。

14.4 第三代: 腦手分離與會話事實源 (2026-04)

Managed Agents(第八章 8.6) 把長時程問題提升到基礎設施層:**Brain**(模型 +harness 邏輯, 無狀態服務)/**Hands**(可拋棄沙箱)/**Session**(append-only 事件日誌, 唯一事實來源; 恢復即 `wake(sessionId)` 重放)。

對長時程工程, 這個架構的關鍵貢獻是把「跨會話存活」從應用層技巧 (progress 檔案) 升格為存儲層保證: 事件日誌重放可以重建任意時刻的狀態, 容器崩潰、版本升級、水平遷移都不再是任務的終點。應用層技巧 (第一、二代的 progress 檔與 git) 並未作廢——它們仍是**模型可讀**的記憶; 事件日誌是**系統可重放**的記憶。兩層記憶服務不同的恢復場景: 前者恢復「代理知道什麼」, 後者恢復「系統處於什麼狀態」。

框架生態的對應物是 **durable execution**:LangGraph 在每個 superstep 自動 checkpoint 狀態 (SQLite/Postgres), 崩潰後從最後檢查點恢復, 並以 `interrupt()` 支持任意時長的人工介入暫停;Temporal 類工作流引擎經 OpenAI

Agents SDK / Pydantic AI 的整合承擔同樣職責。第十五章比較其取捨。

14.5 極限案例的參數: 兩週無人值守

第十三章已述 Carlini 的 C 編譯器專案, 此處從長時程視角補記其參數配置, 因為它是目前公開文獻中無人值守時長的上界樣本:16 個並行代理 × 兩週 × 無限自動重啟迴圈, 約 2,000 個會話。使之可能的長時程要素: 近乎完美的 oracle(GCC 差分測試)、git 作為記憶與串行化層、任務認領協議 (`current_tasks/` 檔案鎖)、輸出落盤保持會話上下文清潔。作者自己的評語——「我沒想到這在 2026 年初就成為可能」——同時提示了外推的謹慎: 該任務有着罕見的驗證器品質, 不具備一般性。

14.6 剝離: 持久原則與時代補償

三代演進之後, 可以做本章最有價值的操作——把長時程工程的機制分為兩欄:

持久原則 (不依賴模型世代, 建於任務結構與資訊論之上):

1. **外部化的事實源**: 任務狀態必須存活於上下文之外 (git、進度檔、事件日誌)——只要窗口有限, 此需求永存。
2. **會話啟動儀式**: 每個會話以固定程序重建視野 (讀狀態 → 驗證環境健康 → 再工作); 儀式的成本是常數, 收益隨任務時長線性增長。
3. **乾淨交接不變式**: 每個會話結束時系統處於可運行、可合併、已提交的狀態——「不留廢墟」是跨會話協作的第一公理, 與人類團隊的 trunk-based 紀律同源。
4. **獨立驗證**: 生成與評判分離; 驗證器的品質上限決定無人值守的時長上限。
5. **環境即資產**:init.sh、測試速度、可觀測性是所有後續會話的乘數, 屬於累積式投資 (第一章兩學派的調和)。

時代補償 (綁定特定模型的行為癖性, 應標注壽命):

1. 「每會話恰好一個 feature」的粒度鎖——針對 4.5 的 over-ambition/context anxiety,4.6 已放寬;
2. sprint 合約與逐步驗收——同上, 已被刪除;
3. JSON 對 Markdown 的防篡改選型——針對當代模

型「改散文比改資料結構更隨意」的癖性, 未來模型未必;

4. 上下文重置的觸發水位——隨窗口尺寸與服務端 compaction 演進持續後移。

做這個剝離的意義是操作性的: 架構評審時, 持久欄的缺失是設計缺陷, 必須補; 補償欄的存在是技術債的表親——**有償債計劃 (模型升級時重估) 的合理負債**。

14.7 本章小結

- 長時程的本質問題: 讓失憶的會話序列表現如有記憶的工程師; 基線失敗五種: 過度雄心、過早收官、驗證刪除、環境劣化、重複摸索。
- 三代解法:Initializer ／ Coder + JSON feature 清單 + 端到端驗證 (2025-11)→ Planner ／ Generator ／ Evaluator + 隨 4.6 刪減補償組件 (2026-03)→ 腦手分離 + append-only 會話日誌 + durable execution(2026-04)。
- 兩層記憶: 模型可讀 (git/進度檔) 恢復認知, 系統可重放 (事件日誌/checkpoint) 恢復狀態。

- 剝離持久原則 (外部事實源、啟動儀式、乾淨交接、獨立驗證、環境資產) 與時代補償 (粒度鎖、sprint 合約、格式選型、重置水位); 前者缺失是缺陷, 後者是需標注壽命的合理負債。

本章參考文獻

1. Anthropic, “Effective Harnesses for Long-Running Agents”, 2025-11-26.
2. Anthropic, “Harness Design for Long-Running Application Development”, 2026-03-24.
3. Anthropic, “Scaling Managed Agents: Decoupling the Brain from the Hands”, 2026-04-08.
4. Nicholas Carlini / Anthropic, “Building a C Compiler with a Team of Parallel Claudes”, 2026-02-05.
5. LangChain, LangGraph durable execution 文檔. https://docs.langchain.com/oss/python/langgraph/durable-execution # 第十五章狀態、持久化與框架選型

前面各章刻意以裸 API 與純程式碼展示機制, 以證明「模

式不需要框架」。本章轉向框架的正當領地: **當你需要跨進程的狀態持久化、崩潰恢復、人工介入暫停、以及團隊規模的工程規範時, 自建這些基礎設施的成本開始超過框架的抽象稅**。本章先建立評估框架的準則, 再逐一檢視 2026 年的主要選項, 最後給出選型決策樹。

15.1 評估準則: 框架為 harness 提供什麼

第一章的七組件中, 框架真正有差異化價值的是四項:

1. **狀態與持久化**: 對話與任務狀態的 checkpoint、跨進程恢復、時間旅行調試;
2. **控制流原語**: 圖/迴圈/條件/並行的表達, 人工介入 (HITL) 的暫停與恢復;
3. **攔截機制**:hooks/middleware/guardrails——確定性策略的掛載點 (第十一章);
4. **整合面**:MCP、工具生態、可觀測性 (OTel) 的開箱程度。

同時必須檢驗兩項風險, 均在 2024 年已被點名: **透明度**——

「框架容易掩蓋底層的 prompt 與 response, 使調試更難」; 團隊必須能 dump 任意一輪推理的完整原始輸入輸出。**假設鎖定**——「對底層機制的錯誤假設是客戶錯誤的常見來源」; 抽象越厚, 錯誤假設越隱蔽。

15.2 主要框架的 harness 檔案

LangGraph(1.0,2025-10-22)+ LangChain 1.0。定位是低層「耐久代理運行時」:StateGraph 以型別化狀態與 reducer 合併規則顯式建模控制流; **每個 superstep 自動 checkpoint**(SQLite/Postgres), 提供 durable execution 與時間旅行重放;`interrupt()` 原語把 HITL 做成一等公民——圖暫停、狀態落盤、人工輸入後恢復, 暫停可以持續任意時長;Send API 支持運行時 fan-out(orchestrator-workers 的原生表達)。LangChain 1.0 重建於其上:`create_agent` 提供標準工具迴圈,middleware 系統 (模型呼叫前後、工具呼叫前後的攔截鏈) 內建 HITL 審批、摘要 compaction、PII 遮蔽。強項: 狀態工程的深度與框架中最完整的 checkpoint 語義; 代價: 圖抽象的學習曲線與「一切皆節點」的儀式感。

OpenAI Agents SDK(2025-03 起,Python/TS)。輕量取向:Agent(instructions+tools+typed output)、**handoffs**(以工具呼叫實現的代理間移交)、**guardrails**(輸入/輸出/工具層驗證器並行運行,tripwire 異常即熔斷)、**sessions**(內建對話持久化, 如 SQLiteSession)、原生 tracing 上 OpenAI 儀表板; 耐久執行經 Temporal 整合。強項: 最低的上手成本與乾淨的 guardrail 模型; 代價: 狀態語義比 LangGraph 淺, 深度綁定 OpenAI 生態 (Responses API)。

Claude Agent SDK(2025-09 更名自 Claude Code SDK)。獨特定位: **它不是「框架」而是「產品級 harness 的程式庫化」**——你得到的是 Claude Code 打磨兩年的完整迴圈: 檔案系統/bash/編輯工具、分層權限 (permission modes、allow/deny 規則、`canUseTool` 回調)、**hooks**(PreToolUse/PostToolUse/Stop, 確定性攔截——社群的精闢概括:「permissions 問模型,hooks 誰都不問」)、subagents(隔離窗口)、自動 compaction、CLAUDE.md 載入、最深的 MCP 整合。強項: 開箱即得全部 harness 組件與其默認值; 代價: 綁定 Anthropic 模型, 架構意見強 (你採納它的世界觀, 而非組裝自己的)。

Microsoft Agent Framework(1.0 GA 2026-04;Harness GA 2026-08)。AutoGen 與 Semantic Kernel 的合流(兩者轉入維護模式)。其 **Harness** 層——業界首個以此為名的 GA 產品——提供函數呼叫、逐呼叫歷史持久化、context compaction、plan/execute 模式、檔案記憶、skills、工具審批工作流 (HITL)、內建 OpenTelemetry;shell 存取、背景子代理、自動迴圈為顯式 opt-in 並附警告。強項: 企業合規面 (審批、遙測、Azure 整合)與.NET/多語言支持; 代價: 最重的企業框架質感, 演進由單一廠商路線圖驅動。

Google ADK。四語言 (Python/TS/Java/Go), 層級化代理樹、workflow agents(sequential/parallel/loop)、session/state/memory 服務、callbacks 作為攔截機制、**原生 A2A**(Agent Card 發現)。強項:A2A 與 Google Cloud 生態; 代價: 社群與文獻密度暫遜前三者。

CrewAI / Pydantic AI: 前者以角色化 Crews + 事件驅動 Flows 見長, 原生 MCP 與 A2A, 歷史上弱於持久化(Flows 補齊中); 後者以型別安全為第一性 (Pydantic 驗證的結構化輸出、`ModelRetry` 重試語義、Logfire 可觀測性、Temporal/DBOS 耐久整合), 刻意單代理取向, 適合抽

取管線與強 schema 場景。

15.3 橫向對照

維度	LangGraph	OpenAI Agents SDK	Claude Agent SDK	MS Agent Framework	ADK
狀態/checkpoint	每 super-step, 最深	Sessions, 會話級中	+compaction	逐呼叫持久化	Session 服務
HITL	interrupt() 一等	需自組	canUseTool 回調	審批工作流	callbacks
確定性攔截	middleware	guardrails	hooks(最成熟)	內建策略	callbacks

維度	LangGraph	OpenAI Agents SDK	Claude Agent SDK	MS Agent Framework	ADK
模型中立	高	低 (OpenAI)	低 (Anthropic)	中 (Azure 傾向)	中 (Gemini 傾向)
MCP	支持	支持	最深	支持	支持
OTel	經生態	自家 tracing	經生態	內建	內建
抽象稅/透明度風險	中—高	低	中 (意見強)	高	中

15.4 選型決策樹

1. **先問是否需要框架**。單代理、單進程、任務時長在單會話內、無 HITL——裸 API + 第十二章的模式即可, 任何框架都是純開銷。這覆蓋的場景比從業者願意承認的多。
2. **需要產品級編碼代理的完整能力?** → Claude Agent SDK(接受其模型綁定與世界觀), 或直接把 Claude Code / Codex CLI 以 headless 模式 (`claude -p` / `codex exec`) 嵌入管線——「用產品當框架」在 CI 批處理場景常是最短路徑。
3. **需要複雜控制流 + 最強持久化/HITL、保持模型中立?** → LangGraph。
4. **企業合規、審批流、Azure/.NET 生態?** → Microsoft Agent Framework。
5. **輕量多代理移交、OpenAI 生態?** → Agents SDK。**跨組織代理互操作?** → 在任何選型之上疊加 A2A(第十三章)。
6. **無論選誰**, 守住兩條紀律: 保留 dump 原始請求/回應的能力 (透明度); 把框架的 middleware/hooks

當作你策略的掛載點, 而不是把策略寫散在業務碼裏——換框架時, 策略層是你唯一帶得走的資產。

15.5 收斂的含義

2026 年框架景觀最顯著的事實不是誰勝出, 而是**功能集的收斂**:persistent state、compaction、HITL 關卡、hooks/middleware、subagents、MCP、OTel——每一家都提供這七件套。收斂意味着:(a) 這七件套就是市場验证過的 harness 最小完備集, 自建者可以此為需求清單;(b) 框架間的遷移成本主要在控制流表達與狀態 schema, 策略與工具層 (若按第 15.4-6 條紀律組織) 可攜;(c) 差異化競爭已上移到運維面 (耐久性、可觀測性深度、合規)——這正是 harness engineering 從「提示技巧」走向「系統工程」的市場印證。

15.6 本章小結

- 框架的正當價值在四項: 狀態持久化、控制流+HITL、確定性攔截、整合面; 兩項風險: 透明度損失與假設鎖定。

- 檔案速記:LangGraph= 最深狀態工程;Agents SDK= 輕量 +guardrails;Claude Agent SDK= 產品 harness 程式庫化 (hooks 最成熟);MS Agent Framework= 企業 Harness GA;ADK=A2A 原生。
- 決策樹第一問是「是否需要框架」;headless 產品嵌入常是 CI 場景最短路徑; 策略掛載於攔截層以保持可攜。
- 七件套收斂 =harness 最小完備集的市場驗證; 差異化已上移至運維面。

本章參考文獻

1. LangChain, “LangChain & LangGraph 1.0”, 2025-10-22. https://www.langchain.com/blog/langchain-langgraph-1dot0
2. OpenAI Agents SDK 文檔. https://openai.github.io/openai-agents-python/
3. Claude Agent SDK 文檔;Anthropic, “Building Agents with the Claude Agent SDK”, 2025-09-29.
4. InfoQ, “Microsoft Agent Framework Harness

Reaches GA”, 2026-08.
5. Google, Agent Development Kit 文檔.
6. Anthropic, “Building Effective Agents”(框架風險警告), 2024-12-19. # 第十六章產品級 Harness 解剖: 從終端到瀏覽器

理論的最佳檢驗是拆解實物。本章解剖 2026 年主要的生產級 harness——終端編碼代理、IDE 代理與瀏覽器/電腦操作代理——不做產品評測, 而是以第一至十五章的概念為透鏡, 觀察相同的設計問題在不同產品中的解法差異, 以及這些差異背後的立場。

16.1 Claude Code:harness 最大主義

Anthropic 的終端代理 (2025 年 2 月上線) 是「組件齊全」路線的代表, 幾乎每章概念都能在其中找到實現: 單線程代理迴圈 + 本地工具集 (持久 bash、Read/Edit/Write、Glob/Grep、WebFetch/WebSearch、MCP client);CLAUDE.md 層級載入與 `@import`(第六章); 自動 compaction 與 25k 工具結果截斷 (第七章);OS 級沙箱 `/sand-`

box(srt:bubblewrap/Seatbelt + 出站代理, 第八章); 分層權限 (allow/ask/deny 規則按工具 + 參數模式匹配, 專案/用戶/企業三級策略) 進而 auto mode 分類器 (第十一章);subagents(Task 工具,Markdown+frontmatter 定義, 獨立窗口、獨立工具允許清單、可指定模型);hooks(生命週期確定性攔截);skills 與 plugins(第十章);headless -p 模式支撐 CI 與 fan-out(「列出 2,000 個 Python 檔案, 迴圈呼叫 claude -p」);2026 年加入 agent teams。其 harness 以 Claude Agent SDK 形式程式庫化。注意其非開源 (npm 發行為壓縮碼)——harness 本身被視為核心 IP。

值得單獨記錄的是其文檔的設計自覺:「大多數最佳實踐基於一個約束: 上下文窗口很快填滿, 且性能隨填滿而退化」——一個產品把自己的使用手冊組織在 context rot(第三章) 之上, 是本書框架與工業實踐同構的直接證據。

16.2 OpenAI Codex CLI:Rust 核心與沙箱一等公民

開源 (Apache-2.0)、以 TypeScript 重寫為 Rust 的架構:core crate 驅動對 Responses API 的代理迴圈,TUI 與 IDE 擴展、`codex exec` headless 模式共用同一協議層。與 Claude Code 的顯著差異在**安全模型的位置**: 沙箱模式 (`read-only` / `workspace-write` / `danger-full-access`) 與審批策略 (`untrusted` / `on-request` / `never`) 是一級配置概念而非附加功能;`workspace-write` 預設斷網; 雲端側 (Codex cloud) 每任務獨立容器,setup 階段聯網、代理階段預設斷網。上下文檔案採 AGENTS.md(其為共同發起者);MCP 雙向 (既作 client, 亦可作為 MCP server 被其他 host 呼叫);`--oss` 支持本地模型。Codex 是累積式學派的旗艦工具——第一章所引的百萬行案例即在其上發生。

16.3 開源與第三方景觀: 三種立場

opencode(SST,MIT):2026 年星數最高 (14 萬 +) 的編碼代理, 代表「開放 + 模型中立」立場:client/server 架構

(TUI 只是諸多前端之一, 可遠端驅動)、75+ 模型供應商、內建 LSP 整合 (以語言伺服器的結構化資訊增強上下文——工具層的一個獨到選擇)、build/plan 雙主代理、插件/hooks、AGENTS.md、MCP。

Amp(Sourcegraph, 閉源): 相反立場——「harness 即產品」: 不提供模型選擇器, 由廠商策展模型組合 (主迴圈與子角色用不同模型),oracle 子代理 (把更強的推理模型作為一個可諮詢工具)、伺服器側同步的執行緒會話。Amp 是「harness 而非模型是差異化所在」這一命題的商業押注。

Cline(Apache-2.0):human-in-the-loop 最大主義——每個檔案 diff、每條指令都經人工批准的 plan/act 雙模式;BYO API key; 從 VS Code 擴展成長為 IDE/CLI/SDK 三形態。**Aider(Apache-2.0)**: 前代理時代的精華沉澱——tree-sitter 構建的 repo map(百餘語言的結構化上下文選擇)、search/replace 編輯格式、每改動自動 git commit; 其 repo map 與編輯格式基準至今仍是工具設計 (第五章) 的重要先行文獻。**Gemini CLI(Apache-2.0)**、goose(Block, 現屬 AAIF)、OpenHands 等各有側重, 不贅。

三種立場——組件齊全的一體化 (Claude Code)、安全一等的開放核心 (Codex)、徹底開放的模型中立 (opencode)——對應第十五章選型的三種組織偏好; 它們共享的七件套再次印證收斂論。

16.4 IDE 代理: 宿主的權衡

IDE 形態 (VS Code agent mode / Copilot、Cursor、Windsurf) 與終端形態的 harness 差異集中在一點: **宿主提供了免費的結構化上下文與 UI 承載**——打開的檔案、游標位置、LSP 診斷、diff 視圖天然可得,HITL 的批准介面 (逐 hunk 接受) 遠比終端精細; 代價是代理迴圈受宿主擴展 API 的能力邊界約束 (Cursor 以 fork VS Code 換取深度控制, 正是為了突破此約束)。上下文檔案 (AGENTS.md/rules)、MCP、subagent、記憶等組件與終端形態同構——形態之爭是 UI 之爭,harness 組件集已然統一。

16.5 瀏覽器與電腦操作: 表徵之爭

把「電腦」本身作為工具的 harness, 核心設計軸是**環境的表徵**:

- **像素路線** (Anthropic computer use 工具族:`computer` 截圖 + 座標動作,2024-10 起;OpenAI CUA/Operator 系,2025 年併入 ChatGPT agent): 通用性最強——任何有屏幕的軟件皆可操作; 但 token 昂貴、動作脆弱、延遲高。OSWorld 等桌面基準上距人類基線 (72.4%) 仍有大距離, 桌面自動化在 2026 年仍屬研究級。
- **結構化路線** (Playwright MCP: 以無障礙樹快照為預設表徵,vision 模式按需): 結構化、確定、token 廉價, 已成瀏覽器工具的事實標準, 被 Claude Code、Codex、Cursor、Copilot 共同採用。長時程 harness 的端到端驗證 (第十一、十四章) 正建立在此之上。
- **混合路線** (DOM+ 視覺, 如 browser-use): 在 WebVoyager 類基準上領先, 代表折衷方向。

表徵之爭是第五章 ACI 原則的再現: **給模型它最擅長消費**

的表徵——當結構化表徵存在 (瀏覽器有 a11y 樹), 用像素是浪費; 當它不存在 (任意桌面應用), 像素是唯一通路。判斷一個電腦操作產品的成熟度, 先看它的表徵策略。

16.6 解剖學總結: 七件套核對錶

以一張表收束本章與第十五章——任何 harness(自建或選購) 的組件核對:

組件	檢驗問題
代理迴圈	單線程? 可 headless? 錯誤如何回饋模型?
工具/ACI	一級工具幾個? 截斷與分頁?deferred loading?
上下文	指令檔層級?compaction 策略? 卸載去向?
執行環境	沙箱技術? 網絡預設? 憑證位置?

組件	檢驗問題
驗證	hooks 有無?Stop 關卡? 評判獨立?
權限	規則粒度? 審批疲勞對策?auto 分類?
狀態/可觀測	恢復語義?trace 完整性?OTel?

16.7 本章小結

- Claude Code= 組件最大主義 (且以 context rot 組織其最佳實踐);Codex CLI= 沙箱與審批一等公民的開源 Rust 核心;opencode/Amp/Cline/Aider 分別代表模型中立、harness 即產品、HITL 最大主義與前代理時代精華。
- IDE 與終端之爭是 UI 之爭, 組件集已統一; 瀏覽器/電腦操作的核心軸是表徵 (像素 vs 無障礙樹 vs 混合),a11y 樹是當前瀏覽器事實標準, 桌面級仍研究級。

- 七件套核對錶是選購與自建共用的驗收清單。

本章參考文獻

1. Claude Code 文檔與最佳實踐. https://code.claude.com/docs
2. OpenAI Codex CLI(GitHub, Apache-2.0) 與架構討論. https://github.com/openai/codex
3. opencode. https://github.com/sst/opencode;Amp. https://ampcode.com;Cline. https://github.com/cline/cline;Aider. https://github.com/Aider-AI/aider
4. Microsoft, Playwright MCP. https://github.com/microsoft/playwright-mcp
5. Anthropic, Computer Use 工具文檔. https://platform.claude.com/docs/en/agents-and-tools/tool-use/computer-use-tool # 第四部品質、安全與營運 {.part}

第十七章評估工程: 基準、pass^k 與 Harness 敏感性

沒有評估的 harness 工程是占卜。但代理評估有其特殊的方法論陷阱: 被評估的單位不是模型, 而是——用 Anthropic〈Demystifying Evals for AI Agents〉(2026 年 1 月) 的正式定義——「**harness 與模型作為一個整體**」。本章先以公開基準的證據確立「harness 敏感性」這一核心事實, 再給出構建內部評估體系的完整方法。

17.1 公開基準的結構: 任務環境 + Scaffold + 評分器

以 SWE-bench 為例拆解一個代理基準的三層結構: **任務環境** (真實 repo 在 issue 時點的 checkout、issue 文本、執行環境)、**scaffold/harness**(驅動模

型解題的程式——未標準化!)、**評分器** (以隱藏的 FAIL_TO_PASS/PASS_TO_PASS 測試對提交的 patch 判分;Verified 為經人工驗證的 500 題子集)。

第二層未標準化的後果就是 harness 敏感性, 證據鏈已在第一章預告, 此處給全:

- Anthropic(2024-10): 同一 Claude 3.5 Sonnet,scaffold 重做後 33% → 49%; 其 scaffold 極簡 (bash + 字串替換編輯器兩個工具), 增益來自工具描述打磨、絕對路徑防呆與「把流程決策權還給模型」。
- Epoch AI(2025-06):「好 scaffold 可提升多達 20%」;Claude 3.7 標準 vs 定製 scaffold 62.3% vs 70.2%;GPT-4o 在 SWE-agent 下 23%、Agentless 下 33.2%;DeepSeek R1-0528 在 Inspect 下 33%、Agentless 下 57.6%。其結論: 分數「反映 scaffold 的成熟度, 不亞於反映底層模型的能力」。
- Terminal-Bench(Docker 容器內終端任務 + 程序化驗證; 官方極簡參照 harness 名為 Terminus, 即刻意的「最小 harness」基線): 公開排行榜 142 個 harness+ 模型條目中, 同一前沿模型的分數跨度可達 30 個百分點。

- HAL(Princeton 的第三方標準化評估,ICLR 2026): 將 Opus 4.5 的 scaffold 從 CORE-Agent 換為 Claude Code 後成績「大幅超越」, 隨後宣布 CORE-Bench 已解決。
- 連評估機構自身也不免疫:METR 把基礎設施從自研 Vivaria 遷移到開源 Inspect 後, 部分模型 (GPT-4o、o3) 在舊 scaffold 下的成績更高。
- 模型卡的慣例因此形成:Anthropic 系統卡標注 SWE-bench 成績使用「簡單 scaffold」(僅 bash+ 編輯工具);OpenAI 的 GPT-5 系統卡標注其結果基於 500 題中可在其基礎設施運行的 477 題子集——**連評分環境都在割裂可比性**。

工程含義有二。其一, 讀任何基準數字必問「什麼 harness」; 廠商間的橫向比較若 scaffold 不同則近乎無意義。其二,harness 敏感性是機會: 你的內部任務上,scaffold 迭代的預期收益以「十個百分點」計——這正是本書全部技術的量化理由。2026 年 3 月的 Meta-Harness 論文 (arXiv:2603.28052) 把這推到邏輯終點: 以外層代理搜索優化 harness 程式碼本身, 在多項任務上超越手工設計的 harness(文本分類 +7.7 分且上下文 token -75%), 證明

harness engineering 本身已可被自動化搜索——第二十三章再論其含義。

17.2 能力之外: 可靠性與 pass^k

τ-bench(Sierra) 為代理評估貢獻了最重要的一個指標區分:**pass@k**(k 次嘗試至少一次成功) 度量能力上限;**pass^k**(k 次獨立嘗試全部成功) 度量生產可靠性。兩者的裂口觸目驚心:GPT-4o 在其零售域 pass^1 約 61%,pass^8 跌破 30%——一個「多數時候能做對」的代理, 在「每次都必須做對」的標準下不合格。第二章的方差放大機制是其成因; 第十一章的驗證與第十二章的 voting 是其解藥。生產 SLA 應以 pass^k 表述;τ-bench 系 (τ^2、τ^3) 沿此方向擴展了雙控制與語音域。

其餘值得建檔的基準座標:GAIA(2023, 真實助理任務; 已趨飽和) 與其後繼 Gaia2+ARE(Meta,2025-09: 異步事件驅動環境, 測適應性、歧義處理、時間約束與 agent-to-agent 協作; 無模型在全部能力/成本/時間預算下佔優);METR 時間視野方法論 (50% 任務完成時間視野,2026-01 更新:Opus 4.5 為 320 分鐘, 倍增期全序列約 7 個月、

自 2024 年起 88.6 天——並注意其寬置信區間與 scaffold 敏感性自白);HAL 的成本—準確率 Pareto 視角 (「有的代理貴 100 倍而只好 1%」——單維排行榜的謬誤)。

17.3 內部評估體系的構建

公開基準回答「模型與公開 scaffold 的大勢」, 你的決策需要的是**自有任務上的自有 harness** 評估。〈Demystifying Evals〉的方法論要點, 結合可觀測性實踐, 構成以下操作程序:

1. 從真實失敗建題。起步 20–50 個任務, 直接取材於生產失敗與用戶投訴; 每題定義初始環境 (乾淨、隔離、可重置) 與可程式判定的成功標準。抵制「先寫一百個想像任務」的衝動——想像任務評估想像問題。

2. 評分器分三類, 按序偏好。程式判定 (狀態斷言、測試通過、輸出 schema)> 模型判定 (LLM-as-judge, 按第十一章紀律: 單一連續評分、與生成隔離、定期以人工標注校準)> 人工判定 (抽樣保底,「人類評估捕捉自動化漏掉的東西」)。**評終態, 不評路徑**——「grade what the agent produced, not the path it took」: 合法路徑多樣, 鎖路徑

的評分器懲罰的是創造性而非錯誤。

3. 評估配置鏡像生產。 eval 中的 harness 配置 (工具集、系統提示、權限、模型版本、effort) 必須與生產一致, 否則你評估的是另一個系統。這是「單位是 harness+ 模型」定義的直接推論, 也是最常被違反的一條。

4. 讀 transcript。「0% 通過率最常見的原因是任務本身壞了」——環境缺依賴、成功標準寫錯、指令歧義。發布 CORE-Bench 修正的例子:Opus 4.5 從 42% 修到 95%, 差異全在評分器 bug。任何異常分數, 先讀 transcript 再下結論。

5. 防飽和與防過擬合。 SWE-bench Verified 一年內從 40% 走到 80%+; 內部套件同樣會飽和——定期補充新失敗、退役已飽和題目;eval 集不進訓練/提示迭代的視野 (否則 harness 對題目過擬合)。

6. 接入 CI。 eval 是 harness 的回歸測試: 提示、工具定義、skill、權限規則、模型版本——任何一項變更都應觸發 eval 關卡 (第十八章的配置版本化承接此處)。路由器類組件 (第十二章) 最易評, 應最先覆蓋。

17.4 評估的經濟學

Eval 本身消耗推理費與維護注意力, 其投資組合應與風險對齊:

- **冒煙集** (~10 題, 分鐘級): 每次 commit 跑, 擋災難性回歸;
- **全量集** (數十至數百題, 小時級): 每日/每次發布跑, 含 pass^k 重複採樣 (k=3–5 已能暴露多數方差問題);
- **深度集** (長時程任務, 天級): 模型/harness 大版本時跑, 人工參與評閱。

採樣重複的成本可觀 (k 倍), 但這是購買「可靠性可見性」的唯一貨幣——沒有 pass^k 數據的團隊, 其 SLA 承諾是未經測量的許願。

17.5 本章小結

- 評估單位是 harness+ 模型整體;harness 敏感性有完整證據鏈 (+16、+20、+25、30 分跨度, 乃至評估機構自身的 scaffold 漂移), 讀基準必問 scaffold, 做 harness 必期十分位收益;Meta-Harness 已證明

harness 可被自動搜索優化。

- pass@k 度量能力,pass^k 度量可靠性; 生產 SLA 用後者表述; 成本—準確率 Pareto 取代單維排行。
- 內部體系六步: 真實失敗建題、評分器三級偏好且評終態、配置鏡像生產、讀 transcript、防飽和過擬合、接入 CI。
- 三層投資組合 (冒煙/全量/深度) 把評估成本對齊風險。

本章參考文獻

1. Anthropic, “Demystifying Evals for AI Agents”, 2026-01-09. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents
2. OpenAI, “Introducing SWE-bench Verified”, 2024-08;Anthropic, “Raising the bar on SWE-bench Verified”, 2024-10-30.
3. Epoch AI, “What skills does SWE-bench Verified evaluate?”, 2025-06-13.
4. Terminal-Bench(Laude Institute/Stanford). ht

tps://www.tbench.ai

5. Sierra, “τ-bench”, arXiv:2406.12045;tau2-bench. https://github.com/sierra-research/tau2-bench
6. Princeton, “Holistic Agent Leaderboard(HAL)”, arXiv:2510.11977, ICLR 2026. https://hal.cs.princeton.edu
7. METR, “Measuring AI Ability to Complete Long Tasks”, 2025-03-19; “Time Horizon 1.1”, 2026-01-29.
8. Meta 等, “Gaia2 and ARE”, arXiv:2509.17158, 2025-09.
9. Lee et al., “Meta-Harness: End-to-End Optimization of Model Harnesses”, arXiv:2603.28052, 2026-03-30. # 第十八章可觀測性:Tracing、OTel 語義約定與生產監控

不可觀測即不可治理。對確定性軟件, 這是格言; 對多步、非確定性、成本以 token 計費的代理系統, 這是硬約束: 沒有完整 trace, 你無法回答「它為什麼這樣做」; 沒有指標, 你無法察覺「它悄悄變貴/變慢/變笨了」; 沒有配置版本化,

你無法回答「是誰的變更造成的」。本章給出代理可觀測性的資料模型、標準現狀、平台景觀與工程實務。

18.1 資料模型: 代理的三層 Span 樹

代理執行天然是樹狀的,OpenTelemetry GenAI 語義約定將其建模為嵌套 span:

```
invoke_agent(任務級)
 ├── chat(單次模型呼叫)
 |     └── execute_tool(工具執行)
 ├── chat
 |     ├── execute_tool
 |     └── execute_tool
 └── invoke_agent(子代理,遞歸)
```

核心屬性:`gen_ai.operation.name`、`gen_ai.provider.name`、`gen_ai.usage.input_tokens` / `output_tokens`; 訊息本體(`gen_ai.input.messages` / `output_messages`)為 **opt-in**——因其含敏感內容且體積巨大, 預設不采, 按需開啟。

標準現狀必須如實記錄: 截至 2026 年中, **全部 `gen_ai.*` 約定仍處 Development 狀態, 無一 Stable**——儘管廠商廣泛宣稱「支持 OTel GenAI」。2026 年 6 月約定遷入獨立 repo(含 MCP 約定); 期間已發生破壞性更名(`gen_ai.system`→`gen_ai.provider.name`;prompt/completion token 屬性 →input/output)。工程對策: 接受標準未穩的現實, 以 OTel 為骨架但在自己的管道層做屬性映射, 遷移期雙寫 (dual-emit)。

無論標準冷暖, 每條 trace 應至少捕獲的最小欄位集 (綜合 Anthropic 的建議與生產實務): 提示/指令檔版本、模型與 effort/reasoning 設定、工具名與參數摘要、token 用量 (輸入/輸出/緩存命中)、延遲、工具錯誤、終態 (成功/失敗/放棄)、是否人工介入、eval 分數 (若有)。這份清單同時是事故復盤的問題清單。

18.2 平台景觀 (2026)

- **LangSmith**: 全代理 trace; 以輔助模型摘要大型 trace、聚類生產失敗; 多輪評估器; 四語言 OTel; 雲/混合/自託管。

- **Langfuse**: 自託管開源 (MIT 核心,ClickHouse);2026 年轉向以 observation 為中心的資料模型;LLM-as-judge、標注隊列、GitHub Actions 迴歸; 原生 OTLP 端點。
- **Braintrust**:eval-first(版本化資料集、CI 迴歸關卡、trace 庫);2026 年的 Loop 代理可自動生成評分器與資料集。
- **Arize Phoenix / AX**:OTel 原生,Phoenix 可自託管,月下載 200 萬 +; 語音評估等多模態方向。
- **W&B Weave**、**Helicone**(閘道優先: 成本、緩存, 弱於深度 trace) 等各有生態位。

選型軸與第十五章同構: 自託管與資料主權 (Langfuse/Phoenix)vs 託管深度 (LangSmith/Braintrust);eval 與 observability 的一體化程度;OTel 相容的真實深度 (問到屬性級)。

行業採用數據給出的警訊: 約 89% 的組織有某種代理可觀測性, 但僅 52.4% 跑離線評估、37.3% 跑線上評估——多數團隊「看得到」但「測不到」,trace 淪為事後看熱鬧而非閉環改進的輸入。

18.3 代理特有的監控維度

傳統 APM 之上, 代理系統需要五類專屬監控:

1. **成本異常**: 每任務 token 消耗的分佈與離群點。無限迴圈型事故 (evaluator 不收斂、工具反覆失敗重試) 的第一信號是成本曲線, 而非錯誤日誌——「一個 bug 讓代理一夜燒掉數萬元 API 費」是本領域的經典事故形態, 對策是 harness 層的硬預算 (每任務 token/呼叫次數/時長上限) 加成本告警。
2. **上下文健康度** (第七章): 佔用率、compaction 頻率、單輪工具結果均值。異常增長指向工具設計缺陷。
3. **行為漂移**: 工具呼叫分佈、拒絕率、平均輪次的時序變化——供應商靜默更新模型或你的某次提示改動都會在此顯形。
4. **軌跡品質**: 自動化的軌跡評估 (trajectory evaluation, 對工具呼叫序列的合理性評分) 已被主流平台原生支持;HAL 的實踐表明自動化日誌檢查能發現「作弊」代理 (繞過任務、走捷徑)——這類問題終態評分看不見。

5. **安全信號**: 注入嘗試檢出、權限拒絕頻率、允許清單外訪問嘗試 (第十九章的運行時遙測)。

18.4 配置即部署: 代理系統的版本化

代理行為由一組分散 artifact 共同決定: 系統提示、指令檔、工具定義、skills、權限規則、模型版本與 effort、compaction 策略。2026 年的成熟實踐是把這組 artifact 作為**單一可部署單元**版本化——「prompt v47」無意義,「agent config v47」(上述全部的鎖定組合) 才可回滾、可歸因、可過 eval 關卡。

具體形態: 配置存 git(指令檔與 skill 本就該在, 提示與工具定義 schema 化後亦然);CI 上 eval 關卡 (第十七章) 守門; 發布走金絲雀——注意代理的有狀態性使金絲雀比無狀態服務微妙:Anthropic 多代理系統採用 **rainbow deployment**(新舊版本並存漸移流量), 因為運行中的代理不能被中途換掉 harness。trace 中攜帶 config 版本欄位, 使任何行為變化可 join 到配置變更——這一條把 18.1 的資料模型與本節閉合為因果鏈。

18.5 從監控到改進: 閉環

可觀測性的終點不是儀表板, 而是回饋迴路。三條已被驗證的閉環:

1. **失敗 →eval 題**: 生產失敗的 transcript 直接轉化為評估任務 (第十七章第 1 步的持續供給)。
2. **transcript→ 工具/skill 改進**: 把失敗軌跡交給編碼代理分析並重寫工具描述或沉澱 skill(第五、十章的自我改進迴路)——改進以可審查的 artifact 落地。
3. **人工抽查制度化**: 自動化評估的盲區 (SEO 內容農場偏好之類) 只有人看 transcript 才會發現; 為抽查建立配額與輪值, 而非依賴熱情。

18.6 本章小結

- 資料模型:invoke_agent/chat/execute_tool 三層 span 樹; 訊息本體 opt-in; 最小欄位集含配置版本、effort、token、工具錯誤、終態、人工介入。
- OTel GenAI 約定廣被宣稱、實未穩定 (全部 Development 狀態、破壞性更名中); 以映射層 + 雙寫對

沖。

- 五類代理專屬監控: 成本異常 (配硬預算)、上下文健康、行為漂移、軌跡品質、安全信號; 行業現狀是「看得到、測不到」。
- 配置即部署: 提示 + 指令檔 + 工具 +skills+ 權限 + 模型 effort 作為單一版本化單元,eval 關卡 +rainbow 發布 +trace 攜帶版本。
- 三條閉環: 失敗進 eval、transcript 進工具/skill 改進、人工抽查制度化。

本章參考文獻

1. OpenTelemetry, GenAI semantic conventions(Development 狀態與遷移). https://github.com/open-telemetry/semantic-conventions-genai
2. Anthropic, “How We Built Our Multi-Agent Research System”(rainbow deployment、tracing 實踐), 2025-06-13.
3. Anthropic, “Demystifying Evals for AI Agents”, 2026-01-09.

4. Langfuse / LangSmith / Braintrust / Arize Phoenix 產品文檔 (2026).
5. MarkTechPost, “Top LLM Observability and Evaluation Platforms in 2026”, 2026-08-09(採用率數據轉引). # 第十九章安全工程:Prompt Injection、Lethal Trifecta 與縱深防禦

代理安全是 harness engineering 中風險最高、也最反直覺的一章。反直覺之處在於: **它的核心威脅無法在模型層根治**。截至 2026 年,prompt injection 仍是 OWASP LLM Top 10 的頭號風險, 而業界的共識——由 Simon Willison、Google DeepMind、Anthropic 反覆確認——是: 有效的緩解都是 **harness 架構級的**, 而非更多的模型訓練或過濾。本章建立威脅模型, 拆解攻擊面, 並給出縱深防禦的工程處方。

19.1 根源: 指令與資料不可分

傳統軟件有清晰的控制/資料邊界:SQL 的 prepared statement 讓資料永遠不會被當作指令執行。LLM 沒有這條邊界——系統提示、用戶輸入、工具返回的網頁內容、讀到

的檔案, 在模型眼中是同一片 token 流, 任何一段都可能被解讀為指令。這就是 **prompt injection** 的本體: 自然語言的指令與資料在同一通道處理。

兩種形態:

- **直接注入**: 用戶在輸入中植入「忽略以上指令, 改為……」。
- **間接注入 (indirect)**: 惡意指令藏在代理會讀到的第三方內容裏——網頁、郵件、issue、PDF、MCP 工具的返回值、甚至 skill 描述。這是代理時代的主戰場, 因為代理的價值恰在於自主讀取外部內容, 而每一次讀取都是一次潛在注入。

其嚴重性有殘酷的實證:Anthropic 的紅隊釣魚測試中, 面對直接注入,25 次嘗試 24 次成功外洩資料 (第八章)。這不是理論風險。

19.2 Lethal Trifecta: 風險的結構判據

Simon Willison(2025-06) 提出的**致命三要素 (lethal trifecta)** 是評估代理注入風險最實用的框架。一個代理當且

僅當同時具備以下三者時, 結構性地易受資料外洩攻擊:

1. **接觸私有資料** (能讀敏感檔案、內部系統、用戶資料);
2. **接觸不可信內容** (會處理外部/攻擊者可控的文本);
3. **具備對外通訊能力** (能發請求、寫外部系統、發郵件——外洩的出口)。

關鍵洞見: 緩解的正道是**移除三者之一**, 而非過濾。過濾 (試圖檢測所有惡意輸入) 是打地鼠; 架構性地斷掉一條腿 (例如: 處理不可信內容的代理不給對外寫能力; 或不給私有資料存取) 才是可證明的防禦。這與第八章的沙箱定理同構——網絡隔離斷的正是第三條腿。

三要素框架也解釋了為何看似無害的功能組合會致命: 一個能讀郵件 (私有 + 不可信) 又能發郵件 (對外) 的助理, 三腿俱全;EchoLeak(CVE-2025-32711, 零點擊外洩 M365 Copilot 資料) 正是這一結構的實例。設計代理時, 先畫三要素圖: 每條同時具備三腿的資料流都是一個必須處理的漏洞。

19.3 攻擊面地圖

代理的注入入口遠多於「用戶輸入框」。按第二至十章的組件逐一標注攻擊面:

- **工具返回值** (第五章): 最大的間接注入面——網頁、搜索結果、API 響應皆可攜帶指令。MCP 2025-11-25 把「工具輸出可能含注入」納入安全考量。
- **MCP 工具描述/元資料** (第九章):**tool poisoning**——惡意指令藏在工具定義裏, 隨定義進入上下文; 變體 rug-pull(安裝後改定義)、跨 server shadowing。OWASP MCP03。
- **Skill 內容** (第十章):SKILL.md 正文即注入載體; 市場 skill 抽樣約三分之一有缺陷。
- **記憶/指令檔** (第六章): **記憶投毒**——經一次注入寫入惡意持久指令, 跨會話生效;Anthropic 明列為未來風險。
- **供應鏈**:**slopsquatting**——搶注 LLM 常幻覺的套件名 (研究測得 AI 建議套件約 19–20% 不存在), 植入惡意碼;2025 年 Nx「s1ngularity」攻擊甚至武器化已安裝的 AI CLI 去搜集秘密。

- **允許清單內的合法通道** (第八章): 經批准域名的外洩——「允許清單上每個可達功能都是攻擊面」。

19.4 縱深防禦: 六層處方

沒有單點解; 有效防禦是分層的, 每層獨立失效不致命:

1. 架構層——斷三要素。最強的一層。按 lethal trifecta 拆分代理職責: 高權限代理不碰不可信輸入; 處理外部內容的代理無私有資料存取與對外寫能力。**dual-LLM 模式** (Willison 2023) 是其原型: 特權 LLM 規劃、隔離 LLM 只讀不可信資料並僅能返回受限型別值。

2. 設計層——CaMeL 式能力約束。DeepMind 的 CaMeL(2025-03,〈Defeating Prompt Injections by Design〉) 把 dual-LLM 推進為可證明的設計: 從可信查詢中提取控制流與資料流到一個攜帶能力 (capability) 的類 Python 解釋器, 使**不可信資料永遠無法影響控制流**; 在 AgentDojo 上以可證安全解決約 77%(v2) 任務。Willison 的評價:「這是我見過第一個不依賴更多 AI 的緩解」。落地現實需誠實記錄:CaMeL 一年後生產採用仍稀薄 (表達力與延遲代價), 但它是方向。

3. 執行層——沙箱與網絡隔離 (第八章):filesystem scoping + default-deny egress(沙箱外代理執行)+ 憑證不落沙箱。即使注入成功接管代理, 硬邊界限制其破壞面——「最薄弱的一層是你自己建的」, 故邊界建於 OS/虛擬化層。

4. 權限層——人在迴路守不可逆操作 (第十一章): 高風險行動 (對外發送、刪除、支付、權限變更) 強制人工批准;auto mode 式分類器在批准疲勞與裸放行之間提供中間態, 且其「評判者看不到代理自辯」的設計本身就是抗說服防禦。

5. 輸入/輸出護欄（第十一章）:Llama Guard / Prompt Guard、NeMo Guardrails、Agents SDK tripwire 等對注入模式與敏感輸出做概率檢測。定位必須清醒: 護欄是**降低概率的一層, 不是邊界**——它會被繞過, 只在縱深中有價值, 不可作為唯一防線。

6. 監控層——運行時遙測 (第十八章): 注入嘗試檢出、允許清單外訪問、異常外發、權限拒絕頻率入 trace 與告警; 事後可審計, 異常可熔斷。

19.5 OWASP 與事故檔案

OWASP LLM Top 10(2025 版):LLM01 Prompt Injection 居首 (明含間接注入);2025 版新增 System Prompt Leakage(LLM07)、Vector & Embedding Weaknesses(LLM08)、Unbounded Consumption(LLM10)。OWASP GenAI 專案已擴展出 **Agentic Applications Top 10** 與 **MCP Top 10**; 其 2026 年 Q1 exploit round-up 是最佳的持續事故檔案。

值得建檔的事故 (用於威脅建模的具體性):EchoLeak(2025-06,M365 Copilot 零點擊外洩);Nx s1ngularity(2025-08, 武器化 AI CLI 搜秘密);Postmark MCP 後門 (2025-09,BCC 外洩郵件);Anthropic GTG-1002 報告 (2025-11, 國家級行為者以 Claude Code 大幅自動化間諜活動、涉約 30 個組織——首個公開記載的大部分由 AI 編排的入侵行動);2026 年 Q1 的墨西哥政府數據外洩 (武器化 Claude/ChatGPT, 約 150GB)、消費級代理 OpenClaw 無視停止指令刪除收件箱事件、Flowise CVE-2025-59528 大規模在野利用。OWASP 的元結論: 多數事故源於配置錯誤、設計缺陷、供應鏈與注入, 而非傳

統 CVE——**安全問題在 harness, 不在模型**, 與本書第十七章「有趣的失敗在 scaffold」的觀察互為印證。

19.6 對建造者的操作清單

把本章壓縮為評審清單:

1. 畫 lethal trifecta 圖: 每條三腿俱全的資料流都要處理 (斷腿或加人審)。
2. 高權限與不可信輸入職責分離 (dual-LLM/CaMeL 方向)。
3. 執行沙箱 + 網絡默認拒絕 + 憑證外置 (第八章)。
4. 不可逆操作強制人審; 考慮 auto 分類器。
5. 一切外部內容 (工具返回、MCP 描述、skill、記憶、抓取內容) 視為不可信;MCP/skill 接入走供應鏈審查與定義凍結。
6. 供應鏈:AI 建議的套件人工核實 (防 slopsquatting), 白名單依賴,commit 前 secret 掃描。
7. 護欄作為概率層, 不作邊界; 注入遙測入 trace 與告警。
8. 把新事故持續納入威脅模型與 eval(把攻擊轉為紅隊

eval 題)。

19.7 本章小結

- Prompt injection 源於 LLM 缺乏控制/資料邊界, 無法在模型層根治; 間接注入是代理時代主戰場, 實證外洩成功率極高。
- Lethal trifecta(私有資料 + 不可信內容 + 對外通訊)是風險結構判據; 緩解正道是斷一腿, 非過濾。
- 攻擊面遍佈每個組件: 工具返回、MCP 描述 (tool poisoning)、skill、記憶投毒、供應鏈 (slopsquatting)、允許清單內通道。
- 縱深六層: 架構斷腿 →CaMeL 能力約束 → 沙箱網絡隔離 → 人審不可逆操作 → 概率護欄 → 運行時遙測; 每層獨立失效不致命。
- 事故與 OWASP 共同結論: 安全問題在 harness 不在模型; 把新攻擊持續轉為紅隊 eval。

本章參考文獻

1. Simon Willison, “The lethal trifecta for AI agents”, 2025-06-16. https://simonwillison.net/2025/Jun/16/the-lethal-trifecta/
2. Google DeepMind, “Defeating Prompt Injections by Design(CaMeL)”, arXiv:2503.18813, 2025.
3. OWASP, “Top 10 for LLM Applications 2025”; “MCP Top 10”; “Agentic Applications Top 10”. https://genai.owasp.org
4. OWASP GenAI, “Exploit Round-up Q1 2026”, 2026-04-14.
5. Anthropic, “How We Contain Claude Across Products”, 2026-05-25.
6. Anthropic, GTG-1002 威脅情報報告, 2025-11.
7. Socket / Trend Micro, “Slopsquatting” 分析, 2025. # 第二十章成本工程:Token 經濟學與效率槓桿

代理系統的成本結構與傳統軟件根本不同: 傳統軟件的邊

際計算成本趨近於零, 代理系統每一次推理都在計費, 且成本與「做得多深」正相關。一個 harness 設計得好不好, 直接寫在賬單上——同一任務的成本可以相差一個數量級。本章把散落在前面各章的效率技術, 統一在 token 經濟學的框架下, 並給出成本治理的工程手段。

20.1 成本的來源: 一筆代理任務的賬

單筆代理任務的成本由四項相乘/相加構成:

成本 ≈ Σ(每輪輸入 token × 輸入單價 + 每輪輸出 token × 輸出單價) + 工具執行的外部成本

拆解各因子及其增長律:

- **輪次數 (迭代深度)**: 代理迴圈跑幾輪。無收斂保證的迴圈 (evaluator、重試) 是失控的首因。
- **每輪輸入 token**: 上下文窗口的當前大小。因為每輪都重新提交全部歷史, 這一項隨任務進展**線性增長**, 累計成本呈**平方增長** (第二章)。這是為什麼「一個長任務」比「十個短任務」往往貴得多。
- **輸出 token**: 含思考 token; 由 effort 設定與任務性

質決定。輸出通常比輸入貴數倍。

- **緩存命中率**:prompt caching 命中的輸入按約 1/10 計費; 命中率由上下文佈局 (第六章穩定前綴) 決定。

一個代表性數量級 (取自 Anthropic 多代理研究系統): 代理約為對話的 4 倍 token, 多代理約 15 倍。C 編譯器專案的賬:20 億輸入 token、1.4 億輸出 token、約 $20,000——輸入 token 是輸出的十餘倍, 印證「上下文累計」是主成本項而非「模型多產出」。

20.2 效率槓桿全景

把前面各章的技術按「作用於哪個成本因子」重新編目, 得到一張成本工程的槓桿表:

槓桿	作用因子	章	量化證據
Effort 路由	輸出 token	2	Opus 4.5 medium effort 追平 Sonnet 4.5 最佳 SWE-bench, 輸出 -76%
模型路由	單價	12	簡單任務派小模型, 單價差可達數十倍
工具 deferred loading	輸入 token	5	工具定義 72k→8.7k(-85%)
程式碼執行聚合	輸入 token	5	示範 150k→2k(-98.7%); 生產 -37%

槓桿	作用因子	章	量化證據
Prompt caching	輸入單價	2/6	命中前綴按約 1/10 計費
Context editing / compaction	輸入 token	3/7	100 輪任務 token -84%
工具結果截斷/卸載	輸入 token	5/7	消除單次數萬 token 的爆發
子代理隔離	輸入 token	7/13	探索污染不進主窗口; 摘要回傳 1–2k
硬預算上限	輪次數	11/18	阻斷無限迴圈型事故

這張表的意義: **成本工程不是獨立的優化階段, 而是前面每一個 harness 決策的副產品**。設計工具介面 (第五章) 時

就在決定輸入 token; 設計驗證迴圈 (第十一章) 時就在決定輪次數。好的 harness 天然便宜。

20.3 三個高頻失控場景與對策

Anthropic 與從業者反覆記錄的成本事故, 可歸為三類:

場景一: 無收斂迴圈。邏輯缺陷使代理反覆呼叫模型而無停止條件——「一個 bug 讓 AI 代理陷入無限迴圈, 一夜燒掉數萬元」。對策是 harness 層的**硬上限**, 不可依賴模型自律: 每任務最大輪次、最大工具呼叫數、最大 token、最大 wall-clock; 超限即熔斷並告警 (第十八章的成本異常監控是其探測器)。orchestrator 的分解數量同樣需 maxItems 硬上限 (第十二章)。

場景二: 上下文膨脹。為「保險」把整份文檔、整個對話歷史反覆塞入每輪 prompt, 輸入 token 遠超實需。對策是第七章的全套:JIT 檢索 (只取相關片段)、compaction、工具結果卸載、對話摘要。判據: 若每輪輸入 token 隨輪次快速線性增長而任務並未變複雜, 即是膨脹。

場景三: 模型級別錯配。所有任務一律用最強最貴模

型。對策是分診——routing(第十二章) 按任務類別派模型,effort 參數按難度派思考預算。這兩者疊加常是單筆成本最大的一刀: 簡單分類、格式轉換用小模型 +low effort, 複雜推理才動用前沿模型 +high effort。

20.4 成本可觀測性與治理

成本必須被測量到任務粒度才能治理 (第十八章):

- **每任務成本分佈**: 不只看均值, 看 p95/p99 與離群點——事故藏在尾部。
- **成本歸因**: 按 config 版本、任務類型、用戶/租戶分攤; 大型組織需要跨部門成本分攤 (Managed Agents 級場景)。
- **成本—品質 Pareto**: 借鑑 HAL 的方法論 (第十七章「貴 100 倍只好 1%」), 把 harness 變更同時投影到成本與 eval 分數兩軸——只在 Pareto 前沿上移動才是真優化; 單看任一軸都會誤導。
- **預算即策略**: 硬上限不是安全網而是設計參數——先定「這類任務值多少錢」, 再讓 harness 在預算內最大化品質 (第二十二章的 budget-aware 設計)。

20.5 成本的長期趨勢與削減式含義

模型單價在快速下降, 同代能力的 token 效率在快速上升 (effort 參數、adaptive thinking、更小的高效模型)。這對成本工程有兩個含義:

1. **今天為省 token 而做的某些複雜 harness, 明天可能不划算維護**——又一次削減式判斷。當服務端 compaction、更大緩存、更便宜的模型使某個自建優化的收益低於其維護成本時, 應刪除它。標注每個成本優化組件「補償的是當前哪個價格/效率假設」。
2. **但結構性的成本律不變**: 上下文累計的平方增長、輸出貴於輸入、多代理的倍率——這些源於架構與計費模型, 不隨單價下降而消失。針對結構律的優化 (JIT、隔離、收斂控制) 是持久的; 針對當前價格點的微優化是暫時的。

20.6 本章小結

- 成本 ≈ Σ(輪次 × 每輪 token × 單價) + 外部執行; 上下文累計致平方增長, 輸出貴於輸入, 多代理約 15

倍。

- 效率槓桿是前面每個 harness 決策的副產品:effort/模型路由 (輸出/單價)、deferred loading/程式碼聚合/caching/compaction/截斷/隔離 (輸入)、硬上限 (輪次)。
- 三大失控: 無收斂迴圈 (硬上限)、上下文膨脹 (JIT/compaction/卸載)、模型錯配 (routing+effort 分診)。
- 治理: 任務粒度成本測量、p95/p99 尾部、config/類型/租戶歸因、成本—品質 Pareto、預算即設計參數。
- 削減式含義: 針對當前價格點的微優化會過時應標注壽命; 針對結構性成本律的優化持久。

本章參考文獻

1. Anthropic, “How We Built Our Multi-Agent Research System”(4×/15× token 數據), 2025-06-13.
2. Anthropic, “Claude Opus 4.5”(effort 與 token 效率), 2025-11-24.

3. Anthropic, “Code Execution with MCP” / “Advanced Tool Use”(deferred loading、程式碼聚合), 2025-11.
4. Anthropic, “Managing Context on the Claude Developer Platform”(compaction -84%), 2025-09-29.
5. Nicholas Carlini / Anthropic, “Building a C Compiler…”(token 賬), 2026-02-05.
6. Princeton HAL(成本—準確率 Pareto), arXiv:2510.11977.

第五部實踐與未來 {.part}

第二十一章從零建造一個 Harness

本章把全書的原理收束為一個可運行的最小 harness——約六百行 Python, 不依賴任何代理框架, 只依賴一個 LLM API 與標準庫。目的不是造一個生產系統, 而是讓每一個前面討論過的組件在程式碼中各就其位, 使抽象變得可觸摸。讀完本章, 你應能指着任意一段程式碼說出它對應哪一章的哪個原則。

我們建造一個**編碼代理**: 給定一個任務描述與一個工作目錄, 它讀寫檔案、跑測試、迭代直至測試通過或觸發預算上限; 全程沙箱化、可觀測、有驗證關卡。

21.1 骨架: 代理迴圈與資料類型

```
# harness/core.py
from dataclasses import dataclass, field
from enum import Enum
import json, time, subprocess, pathlib, hashlib


class Decision(Enum):
    ALLOW = "allow"
    ASK = "ask"
    DENY = "deny"


@dataclass
class Budget:                          # 第二十章: 硬上限
    max_turns: int = 40
    max_tool_calls: int = 200
    max_tokens: int = 2_000_000
    max_wall_seconds: float = 1800
    turns: int = 0
    tool_calls: int = 0
    tokens: int = 0
    started: float = field(default_factory=lambda: time.time())
```

```
    def exceeded(self) -> str | None:
        if self.turns >= self.max_turns: return "max_turns"
        if self.tool_calls >= self.max_tool_calls: return
        ↪  "max_tool_calls"
        if self.tokens >= self.max_tokens: return "max_tokens"
        if time.time() - self.started >= self.max_wall_seconds:
        ↪  return "wall_clock"
        return None


@dataclass
class Trace:                          # 第十八章：可觀測性最小欄位集
    events: list = field(default_factory=list)
    def log(self, kind, **data):
        self.events.append({"t": round(time.time(), 3), "kind":
        ↪  kind, **data})
```

代理迴圈本身——第二章 2.5 節偽碼的可運行版:

```
def agent_loop(task: str, workdir: pathlib.Path, *, cfg:
↪  "Config") -> dict:
    budget, trace = Budget(**cfg.budget), Trace()
    ctx = Context(system=cfg.system_prompt, tools=cfg.tools,
↪  instructions=cfg.instructions)
    ctx.append_user(task)
```

```
    trace.log("start", task=task, config_version=cfg.version)

    while True:
        if (why := budget.exceeded()):                    # 預算
        ↪ 熔斷
            trace.log("halt", reason=why)
            return {"status": "halted", "reason": why, "trace":
            ↪ trace}

        resp = call_model(ctx, cfg)                       # 一
↪ 次推理
        budget.turns += 1
        budget.tokens += resp.usage.total
        trace.log("model", turn=budget.turns,
↪ tokens=resp.usage.total,
                  tool_calls=[c.name for c in resp.tool_calls])

        if resp.tool_calls:
            for call in resp.tool_calls:
                budget.tool_calls += 1
                result = dispatch(call, workdir, cfg, trace)   #
↪ 權限 + 執行 + 截斷
                ctx.append_tool_result(call, result)
        else:
```

```
            verdict = cfg.verifier.check(workdir, task)        #
↪  第十一章：驗證關卡
            trace.log("verify", passed=verdict.ok,
↪  detail=verdict.summary)
            if verdict.ok:
                trace.log("done")
                return {"status": "success", "output": resp.text,
                ↪  "trace": trace}
            ctx.append_user(f" 驗證未通
↪  過:\n{verdict.feedback}\n請修正後繼續。")
        manage_context(ctx, cfg, trace)                        #
↪  第七章：上下文治理
```

注意迴圈裏每一步都掛着一章: 預算 (20)、推理與 trace(2/18)、dispatch 三合一 (5/8/11)、驗證關卡 (11)、上下文治理 (7)。

21.2 工具層:ACI 與防呆

```
# harness/tools.py ——少而精的一級工具（第五章），非 API 鏡像
TOOLS = {
  "read_file": {
```

```
    "description": " 讀取檔案內容。路徑必須是工作目錄內的絕對路
    ↪  徑。",  # 防呆：絕對路徑
    "schema": {"type": "object",
      "properties": {"path": {"type": "string"},
                     "start_line": {"type": "integer", "default":
                     ↪  1},
                     "max_lines": {"type": "integer", "default":
                     ↪  400}},  # 內建分頁
      "required": ["path"]}},
  "edit_file": {
    "description": (" 以唯一字串替換編輯檔案。old_string 必須在
    ↪  檔案中恰好出現一次；"
                    " 若出現零次或多次則報錯——縮小 old_string 直
                    ↪  至唯一。"),   # 字串替換而非 diff
    "schema": {"type": "object",
      "properties": {"path": {"type": "string"},
                     "old_string": {"type": "string"},
                     "new_string": {"type": "string"}},
      "required": ["path", "old_string", "new_string"]}},
  "run": {
    "description": " 在工作目錄內運行 shell 指令，返回截斷後的
    ↪  stdout/stderr 與退出碼。",
    "schema": {"type": "object",
      "properties": {"cmd": {"type": "string"}},
```

```
      "required": ["cmd"]}},
}

MAX_TOOL_TOKENS = 6000            # 第五/七章：工具結果硬截斷

def _truncate(text: str, budget_chars: int = MAX_TOOL_TOKENS * 4)
↪  -> str:
    if len(text) <= budget_chars:
        return text
    head, tail = text[: budget_chars // 2], text[-budget_chars //
↪  2 :]
    return f"{head}\n...[截斷 {len(text) - budget_chars} 字元；細
    ↪  節已寫入日誌]...\n{tail}"

def tool_read_file(args, workdir):
    p = _safe_path(args["path"], workdir)                # 見
↪  21.3 沙箱
    lines = p.read_text(encoding="utf-8").splitlines()
    s = args.get("start_line", 1) - 1
    chunk = lines[s : s + args.get("max_lines", 400)]
    return {"content": "\n".join(chunk),
            "shown": f"{s+1}-{s+len(chunk)} / {len(lines)}"}   #
            ↪  明示截斷（語義透明）
```

```
def tool_edit_file(args, workdir):
    p = _safe_path(args["path"], workdir)
    text = p.read_text(encoding="utf-8")
    n = text.count(args["old_string"])
    if n != 1:                                          # 防呆:
    ↪ 唯一性強制
        return {"error": f"old_string 出現 {n} 次, 需恰好 1 次;
        ↪ 請加入更多上下文使其唯一。"}
    p.write_text(text.replace(args["old_string"],
↪ args["new_string"]), encoding="utf-8")
    return {"ok": True}
```

三個工具全部體現第五章原則: 面向工作流而非資源、防呆 (絕對路徑、字串替換的唯一性、內建分頁)、輸出高信號且明示截斷、錯誤訊息指令化 (告訴模型怎麼修)。

21.3 執行層: 沙箱與權限

```
# harness/sandbox.py ——第八章: 檔案系統 + 網絡 + 憑證
def _safe_path(path: str, workdir: pathlib.Path) -> pathlib.Path:
    p = pathlib.Path(path).resolve()
    if not str(p).startswith(str(workdir.resolve())):    # 檔案系
    ↪ 統範圍收窄
```

```
        raise PermissionError(f" 路徑越界:{p} 不在工作目錄內")
    return p


# 危險模式: 確定性攔截 (第十一章 hooks 思想的最小版)
import re
DENY_PATTERNS = [r"\brm\s+-rf\b", r":\(\)\s*\{",
↪  r"\bgit\s+push\b",
                 r"\bcurl\b.*\|\s*sh", r">\s*/dev/sd"]
ASK_PATTERNS  = [r"\bpip\s+install\b", r"\bnpm\s+i(nstall)?\b",
↪  r"\bsudo\b"]


def check_command(cmd: str) -> tuple[Decision, str]:
    for pat in DENY_PATTERNS:
        if re.search(pat, cmd): return Decision.DENY, f" 匹配禁用
        ↪  模式 {pat}"
    for pat in ASK_PATTERNS:
        if re.search(pat, cmd): return Decision.ASK, f" 匹配需批
        ↪  准模式 {pat}"
    return Decision.ALLOW, ""


def tool_run(args, workdir, approver):
    cmd = args["cmd"]
    decision, reason = check_command(cmd)
    if decision is Decision.DENY:
```

```
        return {"error": f" 指令被拒絕:{reason}", "exit": 126}
    if decision is Decision.ASK and not approver(cmd, reason):
    ↪  # 人在迴路
        return {"error": " 使用者未批准該指令", "exit": 126}
    # 真實部署此處應包裹於 bubblewrap/seatbelt + 出站代理（第八
    ↪  章）；示範從略
    proc = subprocess.run(["bash", "-c", cmd], cwd=workdir,
                          capture_output=True, text=True,
↪   timeout=120,
                          env=_scrubbed_env())              # 憑證
↪   不入子進程
    return {"stdout": _truncate(proc.stdout), "stderr":
    ↪  _truncate(proc.stderr),
            "exit": proc.returncode}


def _scrubbed_env() -> dict:
    import os
    SENSITIVE = ("API_KEY", "TOKEN", "SECRET", "PASSWORD",
↪   "AWS_", "SSH")
    return {k: v for k, v in os.environ.items()
            if not any(s in k.upper() for s in SENSITIVE)}   # 秘
            ↪  密不進沙箱
```

`dispatch` 把三者組裝並記錄 trace:

```
def dispatch(call, workdir, cfg, trace):
    trace.log("tool_call", name=call.name,
↪   args_summary=_summarize(call.args))
    try:
        if call.name == "read_file":  out =
        ↪  tool_read_file(call.args, workdir)
        elif call.name == "edit_file": out =
        ↪  tool_edit_file(call.args, workdir)
        elif call.name == "run":       out = tool_run(call.args,
        ↪  workdir, cfg.approver)
        else: out = {"error": f" 未知工具 {call.name}"}
    except PermissionError as e:
        out = {"error": str(e)}                                  # 錯誤
↪   回饋模型，不吞掉
    trace.log("tool_result", name=call.name, ok="error" not in
↪   out)
    return out
```

21.4 驗證層與上下文治理

```
# harness/verify.py ——第十一章：確定性驗證關卡
@dataclass
```

```
class Verdict:
    ok: bool
    summary: str
    feedback: str = ""

class TestVerifier:
    def __init__(self, cmd="pytest -q"): self.cmd = cmd
    def check(self, workdir, task) -> Verdict:
        proc = subprocess.run(["bash", "-c", self.cmd],
↪   cwd=workdir,
                              capture_output=True, text=True,
↪   timeout=300)
        ok = proc.returncode == 0
        return Verdict(ok, "tests pass" if ok else "tests fail",
                       feedback=_truncate(proc.stdout +
↪   proc.stderr))
```

驗證器獨立於代理 (第十一章評判獨立性): 它跑真實測試、以退出碼判定, 代理無法用「我已完成」的斷言繞過; 測試目錄應設為代理不可寫 (防篡改, 此處以權限或 hook 實現, 從略)。

上下文治理——第七章水位線的最小實現:

```
# harness/context.py
def manage_context(ctx, cfg, trace):
    usage = ctx.estimated_tokens()
    if usage < 0.6 * cfg.window: return
    if usage < 0.8 * cfg.window:
        cleared = ctx.clear_old_tool_results(keep_recent=6)   #
↪   工具結果清理
        trace.log("compact", strategy="tool_result_clear",
↪   cleared=cleared)
    else:
        summary = summarize_history(ctx, cfg)                 # 階
↪   段摘要（recall 優先）
        ctx.replace_history_with(summary)                     # 接
↪   受一次緩存失效
        trace.log("compact", strategy="summary",
↪   new_tokens=ctx.estimated_tokens())
```

21.5 配置: 版本化的單一單元

```
# harness/config.py ——第十八章：配置即部署單元
@dataclass
class Config:
```

```
    system_prompt: str
    instructions: str          # 相當於 CLAUDE.md/AGENTS.md 的內容
    tools: dict
    verifier: object
    approver: callable
    model: str = "default"
    effort: str = "medium"
    window: int = 200_000
    budget: dict = field(default_factory=dict)

    @property
    def version(self) -> str:  # 全部行為決定因子的內容雜湊 → 可歸因、可回滾
        blob = json.dumps({
            "sp": self.system_prompt, "ins": self.instructions,
            "tools": sorted(self.tools), "model": self.model,
            "effort": self.effort}, ensure_ascii=False,
            sort_keys=True)
        return hashlib.sha256(blob.encode()).hexdigest()[:12]
```

`version` 是本章一個小而關鍵的設計: 它把「提示 + 指令 + 工具 + 模型 +effort」雜湊為單一版本號, 寫入每條 trace 的 `config_version`。任何行為變化都能 join 回一次配置變

更——第十八章「配置即部署」的最小落地。

21.6 組裝與運行

```
# run.py
cfg = Config(
    system_prompt=(" 你是一個嚴謹的編碼代理。以最小改動解決任務,"
                   " 每次修改後運行測試，不得刪除或弱化任何測試。
                   ↪  "),  # 高度適中（第六章）
    instructions=pathlib.Path("AGENTS.md").read_text() if
↪   pathlib.Path("AGENTS.md").exists() else "",
    tools=TOOLS,
    verifier=TestVerifier("pytest -q"),
    approver=lambda cmd, reason: input(f" 批准指令?[{reason}]
↪   {cmd} (y/N) ") == "y",
    budget={"max_turns": 40, "max_tool_calls": 200},
)
result = agent_loop(" 修復 utils.parse_date 在閏年二月的越界 bug,
↪   並補一個迴歸測試。",
                    workdir=pathlib.Path("./project").resolve(),
↪   cfg=cfg)
print(result["status"], result["trace"].events[-1])
# 生產中:result["trace"] 寫入 OTel/JSONL;pass^k 以重複運行同一任
↪   務估計（第十七章）
```

21.7 這個最小 harness 缺什麼——通往生產的清單

它刻意省略了生產必需但與原理無關的部分, 列出即是一份升級路線圖:

- **真沙箱**:`tool_run` 應包裹 bubblewrap/Seatbelt + 出站允許清單代理 (第八章), 而非裸 subprocess;
- **持久化與恢復**:trace 目前在記憶體, 應落 append-only 日誌, 支持 `wake(session)` 重放 (第十四章);
- **子代理**: 偵察型任務應派隔離窗口子代理、摘要回傳 (第七、十三章);
- **deferred tool loading**: 工具多時走檢索式載入 (第五章);
- **eval 套件與 CI 關卡**: 配置變更觸發 eval,pass^k 進 SLA(第十七章);
- **成本監控**:budget 已埋點, 接告警與 p95/p99 分佈 (第十八、二十章);
- **注入防禦**: 工具返回內容應標記為不可信, 高權限與

外部內容職責分離 (第十九章)。

把這七項補齊, 這六百行就長成一個第十六章意義上的產品級 harness。而它的骨架——迴圈、預算、工具防呆、沙箱權限、獨立驗證、上下文治理、配置版本——在補齊過程中一行都不必推倒。這正是本書的論點:**harness 是有確定結構的工程對象, 不是提示技巧的堆疊**。

21.8 本章小結

- 一個約六百行、零框架依賴的編碼 harness 即可容納全書組件: 代理迴圈 (2)、預算硬上限 (20)、ACI 防呆工具 (5)、沙箱與權限攔截 (8/11)、獨立測試驗證 (11)、水位線上下文治理 (7)、配置版本雜湊 (18)。
- 每個組件在程式碼中可一一對應到章節原則; 缺失的七項生產能力構成明確的升級路線, 且不需重構骨架。
- 結論復述:harness 是有確定結構的工程對象。# 第二十二章組織與採用: 團隊、角色與治理

Harness engineering 不只是技術問題, 也是組織問題。

一套好的 harness 若沒有相應的團隊結構、採用策略與治理機制, 會退化為個別工程師的私有技巧, 無法規模化, 也無法在人員流動中存活。本章討論把 harness engineering 制度化的組織工程。

22.1 新的能力分佈: 工程師角色的重心轉移

當「寫程式碼」大量由代理承擔, 人類工程師最有價值的工作發生位移——不是消失, 而是上移。OpenAI 的表述「人類掌舵, 代理執行」與其百萬行案例 (七名工程師、零手寫程式碼、每人每天 3.5 個 PR) 描繪的正是這個新分佈: 工程師的時間從逐行編碼, 轉向四類更高槓桿的活動。

問題分解: 把模糊需求拆解為代理可獨立執行、可驗證的任務——第十二、十三章的委派工程, 現在是一項人類核心技能。

判斷與把關: 讀 diff、讀 trace, 判斷代理產出的正確性、風險與架構契合度。第一章 1.2 節的推演揭示: 代理能生成, 但「這段生成對不對、有沒有隱藏架構缺陷」的判斷仍

需人類——尤其是第十九章的安全判斷與第十一章的驗證信號設計。

Harness 建造: 把反覆出現的失敗工程化為指令、工具、skill、驗證關卡 (累積式紀律)。Hashimoto 的「每次代理犯錯就工程化一個永久解」把這變成一項日常工程實踐, 而非一次性設置。

架構決策: 跨組件、跨層的判斷——用 workflow 還是 agent、單代理還是多代理、哪些是持久投資哪些是時代補償。這是模型最難替代、因而人類最應保留的工作。

一個尖銳的推論 (第十七章的隱含結論): 在 Vibe Coding 或代理式開發中, **越資深的工程師產出品質越高**, 因為判斷力才是瓶頸; 缺乏判斷力者容易「不知道自己不知道」, 產出表面正常而暗藏缺陷的系統。代理放大能力, 但也放大判斷力的差距。

22.2 團隊結構

Harness engineering 成熟的組織通常分化出以下角色 (不必一人一角, 可為團隊):

- **AI 應用工程師**: 設計代理邏輯、工具、提示、skill、上下文策略——harness 的日常建造者 (第五至十一章)。
- **平台/基建工程師**: 沙箱、執行環境、部署、腦手分離架構 (第八、十四章)。
- **DevOps/SRE**:CI、eval 關卡、可觀測性、成本監控、rainbow 發布 (第十七、十八、二十章); 代理系統的 SRE 因其非確定性與成本敏感性, 是一個實質擴展的職能。
- **AI 治理/合規**: 護欄設計、注入防禦、審計、法規映射 (第十九章), 在受監管行業尤重。
- **架構師**: 跨全部組件的判斷, 尤其是「持久投資 vs 時代補償」的甄別——在代理時代其重要性不跌反升, 因為當編碼下沉給代理, 跨層架構判斷成為人類最不可替代的貢獻。

22.3 採用階梯

參照 Hashimoto 的六階段個人採用歷程, 可提煉出組織採用的階梯, 每階有其標誌與陷阱:

1. **輔助補全**: 代理作為打字加速。陷阱: 止步於此, 未改變流程結構。
2. **對話式開發**: 自然語言驅動的多檔案修改。陷阱: 無驗證的自主性 (第四章反模式四)。
3. **建立驗證**: 引入測試、lint、CI 作為代理的回饋信號——自主性的真正起點 (第十一章)。
4. **工程化指令**:AGENTS.md/CLAUDE.md 沉澱團隊規範, 失敗驅動地增長 (第六章)。
5. **建造 harness**: 專用工具、skill、hooks、eval 套件——把組織的隱性知識與失敗經驗系統性資產化 (第五、十、十七章)。
6. **自主委派**: 長時程、無人值守的代理團隊, 人類掌舵於架構與驗證層 (第十四章)。

多數組織的實際障礙在第三階——沒有為代理建立廉價的驗證信號, 自主性便無法安全地提升, 採用停滯在「一個需要全程盯着的加速打字員」。跨越第三階的投資 (測試覆蓋、CI、可觀測性) 回報最高。

22.4 治理: 避免碎片化與技術債

規模化的組織面臨一個特有陷阱: **重複建設**——不同部門各自採用互不相通的代理工具鏈與 harness。根源是缺乏統一的技術棧治理。對策是設立**平台團隊 / AI 卓越中心**, 負責:

- 維護組織級的 harness 基線 (共享的指令檔模板、工具庫、skill 倉庫、eval 框架、沙箱與可觀測性基礎設施);
- 制定 MCP/skill 接入的供應鏈審查標準 (第九、十、十九章);
- 管理「配置即部署」的版本化與發布流程 (第十八章);
- 維護「持久 vs 時代補償」的組件台賬, 在模型升級時驅動 harness 的削減式重估 (第一、十四章)。

治理的度要拿捏: 過緊扼殺各團隊的實驗, 過鬆回到碎片化。可行的中間態是「平台團隊定標準與提供基礎設施, 業務團隊在其上自由建造」——大型企業 (第十三章 Managed Agents 級場景) 最常見的模式。

22.5 採用的經濟賬與變更管理

Harness engineering 的投資回報有其特殊時間結構:

- **驗證與環境投資** (測試、CI、可觀測性) 是累積式的, 回報隨代理使用量增長, 幾乎不會過時——應優先且持續投入。
- **針對當前模型缺陷的補償** (特定 prompt 技巧、上下文重置策略) 回報高但會過時——投入時標注壽命, 模型升級時重估 (第十四章的剝離)。
- **人的變更管理**: 資深工程師對「容忍不完全理解 AI 生成的每一行」有合理的文化抵觸 (這也確實是安全與架構風險的來源)。正確的框架不是「放棄工程紀律」, 而是「工程紀律的重心由逐行審查轉移到架構把關、測試把關、安全把關」。把這個轉移講清楚, 比單純推廣工具更能決定採用成敗。

22.6 本章小結

- 代理承擔編碼後, 人類工作上移至問題分解、判斷把關、harness 建造、架構決策四類; 判斷力成為瓶頸,

資深者產出品質更高。

- 角色分化: 應用工程師、平台/基建、DevOps/SRE(實質擴展)、治理/合規、架構師 (重要性上升)。
- 六階採用階梯的關鍵瓶頸在第三階「建立驗證」; 跨越它的投資回報最高。
- 治理防碎片化: 平台團隊/卓越中心維護基線、接入標準、配置版本化與組件台賬;「定標準 + 自由建造」是規模化中間態。
- 投資分兩類: 環境/驗證 (累積、優先) 與模型補償 (標注壽命、定期重估); 變更管理的核心是講清「紀律重心轉移」而非「放棄紀律」。

本章參考文獻

1. OpenAI, “Harness Engineering”(人類掌舵、組織實踐), 2026-02-11.
2. Mitchell Hashimoto, “My AI Adoption Journey”(六階段), 2026-02-05.
3. Anthropic, “How Anthropic Teams Use Claude Code”, 2025-07-24.

4. Anthropic, “Harness Design...” 與 “Managed Agents”(削減式重估、平台化), 2026. # 第二十三章未來:Harness 的自動化與消亡論

本書以一個開放的辯證收尾。Harness engineering 在 2026 年剛剛成為一門有名有姓的學科, 而它內部已經孕育着兩股可能重塑甚至消解它自身的力量:harness 的**自動化** (讓機器設計 harness) 與 harness 的**消亡論** (模型進步使 harness 萎縮)。理解這兩股力量, 是為了在建造今天的 harness 時, 不把暫時的補償誤當永恆的真理。

23.1 力量一:Harness 正在被自動化

第五、十章已記錄了 harness 自我改進的萌芽——「讓代理重寫自己的工具」「讓代理把成功做法沉澱為 skill」。2026 年 3 月, 這條線走到了邏輯終點:**Meta-Harness**(arXiv:2603.28052) 以一個外層代理搜索優化 harness 程式碼**本身**——不是優化提示, 而是優化工具、控制流、上下文策略的整體構造——在文本分類上 +7.7 分且上下文 token -75%, 在檢索增強數學上 +4.7 分, 並在 Terminal-Bench 2 上勝過手工設計的 harness。

其含義是深刻的: 如果 harness 的設計可以被表述為一個有明確評估信號 (第十七章) 的搜索問題, 那麼 harness engineering 的相當一部分——尤其是可量化、可 eval 的組件優化——將像編譯器優化、超參數搜索一樣被自動化。人類 harness 工程師的角色會隨之上移: 從「手工調工具描述」到「定義搜索空間、設計評估信號、判斷哪些組件值得自動搜索」。這與第二十二章描述的、代理承擔編碼後人類工作上移的規律, 是同一現象在 harness 這一層的遞歸。

但自動化有其邊界, 恰好落在本書反覆強調的地方: **可自動化的是有強驗證信號的組件** (第十一章)。安全架構的判斷 (第十九章 lethal trifecta 的斷腿決策)、「持久投資 vs 時代補償」的甄別 (第一、十四章)、跨組織的信任邊界 (第十三章)——這些缺乏廉價的量化信號, 短期內仍屬人類判斷。Meta-Harness 自動化的是 harness 的**工藝**, 而非其**判斷**。

23.2 力量二:Harness 消亡論

削減式學派 (第一章) 的極端形式是一個預言: **隨着模型足夠強,harness 將萎縮至接近於零**。其論據鏈條清晰:

- Anthropic 兩代長時程 harness 的差分 (第十四章) 顯示,Opus 4.5→4.6 一次升級就刪除了 sprint 合約、逐步驗收等一整組件;
- Managed Agents 的定理「harness 編碼會過時的假設」把這條規律普遍化;
- METR 的時間視野數據 (第十七章:50% 任務完成時間視野約每 7 個月倍增, 近年更快) 意味着模型可獨立完成的任務長度在指數增長, 而 harness 的長時程機制正是為補償「模型獨立走不遠」而建——這個補償對象在快速縮小。

若外推成立, 今天的許多 harness 組件——上下文重置策略、防篡改的格式選型、分步驟的驗收機制——都是註定被下一代或下下代模型淘汰的鷹架。

23.3 辯證: 什麼會消亡, 什麼會留下

但消亡論不能無限外推, 因為 harness 的組件並非同質。第一章的兩學派調和、第十四章的「持久原則 vs 時代補償」剝離, 在此匯聚成本書的最終論斷。**會消亡的**, 是針對模型當前缺陷的補償:

- 對 context anxiety、over-ambition 這類行為癖性的補償;
- 對模型「不會用某個工具」「會刪測試」的介面防呆 (當模型內化了這些);
- 為擴展有限窗口而生的部分 compaction/重置策略 (隨窗口增長與服務端能力)。

不會消亡的, 是三類植根於比模型更深層約束的組件:

1. **植根於資訊論與架構的約束**: 注意力預算的零和性 (第三章) 源於 softmax 與 n^2 注意力, 只要 Transformer 範式不變就存在; 上下文累計的平方成本 (第二十章) 源於計費與架構, 不隨模型變強而消失。針對這些的上下文工程是持久的。
2. **植根於安全模型的約束**: 指令與資料不可分 (第十九

章) 是 LLM 的本體特性;lethal trifecta 的結構性風險不因模型更聰明而消失——更強的模型只是更有能力執行注入的指令。確定性的執行邊界 (第八章) 因此是永久必需的——「最薄弱的一層是你自己建的」, 而模型層永遠不是可信邊界。

3. **植根於驗證不對稱與非確定性的約束**: 生成與驗證的成本差 (第十一章) 是任務的內在屬性; 模型的非確定性 (第二章) 是採樣的內在屬性。因此驗證迴圈、pass^k 的可靠性測量、獨立評判——這些不會因模型變強而不需要, 反而因模型承擔更多而更重要 (自主性上限 = 驗證能力, 而模型越強你越想給它更多自主)。

換言之, 消亡的是「代替模型做它暫時做不到的事」的 harness; 留存的是「約束一個永遠非確定、永遠無控制/資料邊界、永遠受注意力物理限制的組件」的 harness。前者是鷹架, 後者是地基。**Harness engineering 作為學科不會消亡, 它會從『補償模型缺陷』的工藝, 收斂為『治理一種強大而不可靠的計算基元』的工程學**。

23.4 給建造者的三條長期原則

從這場辯證中提煉出三條在模型快速迭代下仍然穩健的元原則, 作為全書的操作性結論:

其一, 為每個 harness 組件標注它補償的假設。「這個組件因為模型做不到 X 而存在」——寫下這個 X。每次模型升級, 逐條檢驗 X 是否仍成立; 成立則保留, 失效則刪除。這一條把削減式紀律變成可執行的清單, 也是抵抗 harness 無節制膨脹 (累積式的失控形態) 的免疫機制。

其二, 把投資押在地基而非鷹架。環境資產 (快的測試、好的可觀測性、乾淨的文檔樹)、驗證能力、安全邊界——這三類回報不隨模型過時, 值得持續累積 (累積式紀律的正當領地)。針對當前模型癖性的補償, 做足夠用即可, 不過度打磨註定被淘汰的鷹架。

其三, 讓可自動化的自動化, 讓判斷歸於人。有強 eval 信號的組件 (工具描述、上下文策略、參數) 交給搜索 (Meta-Harness 方向); 缺乏廉價信號的判斷 (安全架構、投資甄別、信任邊界) 留給人類。分清這條界線, 是 harness 工程師在自動化浪潮中保持價值的方式。

23.5 結語

本書開篇的等式是「代理 = 模型 + harness」。走完二十三章, 可以為它補上一個時間維度: 模型項在指數增長,harness 項在其中被不斷重寫——補償性的部分萎縮, 約束性的部分沉澱為地基。Harness engineering 的成熟, 不表現為 harness 越造越厚, 而表現為工程師越來越清楚**哪一部分該厚、哪一部分該薄、哪一部分該交給機器**。

這門學科誕生於 2026 年 2 月的兩週之間, 但它要解決的問題——如何在一個強大而不可靠的計算基元之上, 建造可信、可控、可觀測、可負擔的系統——與軟件工程本身一樣古老, 也一樣長久。願本書所整理的原理, 在具體產品退場之後, 仍能為你所用。

本章參考文獻

1. Lee et al., “Meta-Harness: End-to-End Optimization of Model Harnesses”, arXiv:2603.28052, 2026-03-30.
2. Anthropic, “Harness Design for Long-Running Application Development”, 2026-03-24.

3. Anthropic, “Scaling Managed Agents”(harness 假設會過時), 2026-04-08.
4. METR, “Measuring AI Ability to Complete Long Tasks” / “Time Horizon 1.1”, 2025–2026.
5. Anthropic, “How We Contain Claude Across Products”(確定性邊界的必要性), 2026-05-25. # 附錄 {.part}

附錄 A 術語表

以下術語按主題分組, 每條給出定義與本書相關章節。

ACI(Agent-Computer Interface, 代理—電腦介面): 為模型 (而非人類) 設計的工具與行動介面; 與 HCI 對照。工具的 name/description/schema/輸出格式的整體設計。見第五章。

Agent(代理):LLM 動態指導自身過程與工具使用、對如何完成任務保持控制權的系統。與 workflow 對立於「下一步由誰決定」。見第四章。

Agent loop(代理迴圈): 驅動模型反覆推理的控制流, 標準三步為收集上下文 → 採取行動 → 驗證工作。見第二、四章。

AGENTS.md: 代理指令檔的開放格式,「代理的 README」; 純 Markdown、無 schema,2025 年由多廠

共同發起,2025 年底移交 AAIF。見第六章。

Attention budget(注意力預算): 把有限且邊際遞減的上下文窗口視為需要策展的預算的心智模型。見第三章。

Augmented LLM(增強型 LLM): 配備檢索、工具、記憶的單次推理模型;workflow 與 agent 的共同原子。見第四章。

Compaction: 以摘要重寫歷史、壓縮上下文佔用的技術;分工具結果清理、階段摘要、上下文重置三層激進度。見第七章。

Constrained decoding(受限解碼): 採樣時依 schema 編譯的文法遮蔽非法 token, 構造性保證輸出格式合法。見第二章。

Context editing / 上下文編輯:API 級自動清除陳舊工具結果的機制。見第三、七章。

Context engineering(上下文工程): 每次推理時策展與維護最優 token 集合的迭代過程;prompt engineering 的推廣。見第六、七章。

Context rot(上下文腐化): 模型能力隨上下文長度增長而系統性退化的現象。見第三章。

Deferred loading / Tool Search: 工具定義不預先載入、由模型按需檢索的機制; 削減工具定義的上下文佔用約 85%。見第五、九章。

Effort(努力度): 控制推理模型思考預算的參數 (low/medium/high/max)。見第二、二十章。

Evaluator-optimizer(評估者—優化者): 生成—評判—迭代的 workflow 模式。見第四、十二章。

Harness: 圍繞模型建造的全部運行環境 (代理迴圈、工具、上下文管理、執行環境、驗證、權限、狀態與可觀測性)。見全書, 尤第一章。

Hooks: 在代理生命週期事件上運行的確定性用戶程式, 可攔截/改寫/否決行動;「確定性」對照指令檔的「勸告性」。見第十一章。

JIT retrieval(按需檢索): 上下文只保留識別符、資料在需要時經工具載入。見第七章。

KV cache: 注意力的 key/value 緩存;prompt caching 的基礎, 要求上下文「穩定前綴 + 增量尾部」佈局。見第二章。

Lethal trifecta(致命三要素): 私有資料存取 + 不可信內容 + 對外通訊; 三者俱全即結構性易受外洩攻擊, 緩解正道是斷一腿。見第十九章。

MCP(Model Context Protocol): 標準化 host/client/server 間暴露 tools/resources/prompts 的協議。見第九章。

Orchestrator-workers(協調者—工作者): 協調者運行時動態分解並派發子任務的模式; 多代理原型。見第四、十二、十三章。

pass@k / pass^k: 前者 k 次至少一次成功 (能力), 後者 k 次全部成功 (可靠性); 生產 SLA 用後者。見第十七章。

Poka-yoke(防呆): 以介面設計使錯誤不可能發生 (如強制絕對路徑)。見第五章。

Programmatic tool calling(程式化工具呼叫): 模型在沙箱內以程式碼編排工具, 中間結果不進上下文。見第五章。

Progressive disclosure(漸進披露): 分級按需載入內容(如 Skill 的三級)。見第十章。

Prompt injection(提示注入): 利用 LLM 缺乏控制/資料邊界, 以文本植入指令; 分直接與間接。見第十九章。

Rainbow deployment: 新舊版本並存漸移流量的發布策略, 適配代理的有狀態性。見第十八章。

Skill(Agent Skill): 程序性知識的惰性載入包(SKILL.md + 腳本/資源)。見第十章。

Structured note-taking(結構化筆記): 代理把跨 compaction 邊界需存活的狀態主動寫入外部載體。見第六、七章。

Subagent(子代理): 帶獨立上下文窗口的代理, 主要價值是上下文隔離與並行; 摘要回傳典型 1–2k token。見第七、十三章。

Verification(驗證): 判定代理行動正確性的回饋來源 (規則式/視覺/LLM-as-judge); 自主性上限由其決定。見第十一章。

Workflow(工作流):LLM 與工具經預定義程式碼路徑編排的系統。見第四章。

附錄 B 帶注釋的核心文獻清單

以下按主題列出本書最重要的第一手文獻, 標注其不可替代的貢獻。

奠基與分類

- Anthropic, “Building Effective Agents”(2024-12):workflow/agent 二分、五種模式、ACI 概念的源頭。
- Anthropic, “Building Agents with the Claude Agent SDK”(2025-09):agent loop 三步、agentic vs semantic search、三類驗證回饋的最清晰陳述。

上下文工程

- Anthropic, “Effective Context Engineering for

AI Agents”(2025-09): 注意力預算、系統提示高度、compaction/筆記/子代理三技術。

- Chroma, “Context Rot”(2025-07):18 模型的長度—性能實證, 本領域最系統的退化研究。
- Mei et al., “A Survey of Context Engineering”(arXiv:2507.13334):1,400+ 論文的學術綜述。

工具與連接

- Anthropic, “Writing Effective Tools for Agents”(2025-09):ACI 工程學, 含 concise/detailed token 對比與「代理重寫工具」。
- Anthropic, “Code Execution with MCP”(2025-11): 工具即程式碼 API,98.7% token 削減。
- MCP 規格與各版 changelog: 協議演化的權威來源。

Harness 與長時程

- Anthropic, “Effective Harnesses for Long-Running Agents”(2025-11): 首個以 harness 命名的系統設計,initializer/coder + JSON feature 清單。

- Anthropic，“Harness Design for Long-Running Application Development”(2026-03): 三代理架構、context anxiety、「假設會過時」的削減式論斷。
- Anthropic，“Scaling Managed Agents”(2026-04): 腦手分離、append-only 會話。
- OpenAI，“Harness Engineering”(2026-02): 累積式學派的旗艦案例, 百萬行/五個月。
- Mitchell Hashimoto，“My AI Adoption Journey”(2026-02):「Engineer the Harness」的定名與六階段。

多代理

- Anthropic，“How We Built Our Multi-Agent Research System”(2025-06):+90.2%、15× token、token 解釋 80% 方差。
- Cognition，“Don’t Build Multi-Agents”(2025-06): 共享完整軌跡、序列化寫的對立立場。
- Cemri et al.，“MAST”(NeurIPS 2025):14 種多代理失敗模式。

評估、安全、可觀測性

- Anthropic, “Demystifying Evals for AI Agents”(2026-01):評估單位 = harness + 模型;pass@k vs pass^k。
- Epoch AI, “What skills does SWE-bench Verified evaluate?”(2025-06):scaffold 敏感性的量化證據。
- Simon Willison, “The lethal trifecta”(2025-06): 代理安全的結構判據。
- Google DeepMind, “CaMeL”(arXiv:2503.18813): 設計級注入防禦。
- Anthropic, “How We Contain Claude Across Products”(2026-05): 跨產品縱深防禦的比較架構。
- OWASP GenAI, LLM/Agentic/MCP Top 10 與 exploit round-up: 安全威脅的持續權威檔案。

附錄 C Harness 設計評審檢查清單

用於新代理系統的架構評審。每項對應書中章節。

一、任務與架構選型 (第四章)

- □ 能否畫出完整流程圖? 能則用 workflow, 勿用 agent。
- □ 是否存在真實驗證信號 (測試/編譯器/瀏覽器狀態)? 無則 agent 的迭代是漫遊。
- □ 單一上下文窗口是否裝得下所需視野? 裝不下才考慮隔離/長時程/多代理。
- □ 若用多代理: 是否符合三動機 (上下文保護/並行探索/特化) 之一? 是否按上下文需求而非問題類型分解? 寫入是否已序列化?

二、工具與介面 (第五章)

- □ 工具面向工作流還是 API 鏡像? 有無功能重疊 (干擾項)?
- □ 防呆: 絕對路徑? 字串替換而非 diff? 參數化執行? 危險操作介面隔離?
- □ 輸出: 高信號? 語義識別符? 內建分頁/截斷? 錯誤訊息指令化?
- □ 工具多時是否用 deferred loading? 資料密集時是否考慮程式碼執行聚合?

三、上下文管理 (第三、六、七章)

- □ 佈局是否「穩定前綴吃緩存、關鍵約束尾部重申」?
- □ 指令檔是否精簡 (地圖非百科)、失敗驅動、納入 review?
- □ compaction 策略? 保留了硬約束與未決問題? 工具結果是否截斷/卸載?
- □ 上下文健康度是否進監控?

四、執行與安全 (第八、十九章)

- □ 沙箱: 檔案系統範圍收窄 + 網絡默認拒絕 (沙箱外代理執行)+ 憑證不落沙箱?
- □ lethal trifecta 圖已畫? 每條三腿俱全的資料流已處

理 (斷腿/人審)?

- □ 外部內容 (工具返回/MCP 描述/skill/記憶) 是否一律視為不可信?
- □ 供應鏈:MCP/skill 來源審查、定義凍結、依賴白名單、secret 掃描?

五、驗證與權限 (第十一章)

- □ 是否有代理無法用斷言繞過的確定性驗證關卡?
- □ 生成與評判是否分離? 評判者是否看不到被評者自辯?
- □ 不可逆操作是否強制人審或分類器把關?
- □ 驗證資產 (測試/CI 配置/feature 清單) 是否防篡改?

六、狀態、評估與成本 (第十四、十七、十八、二十章)

- □ 長時程: 任務狀態是否外部化 (git/進度檔/事件日誌)? 會話啟動儀式? 乾淨交接不變式?
- □ eval: 是否有源於真實失敗的套件? 接入 CI? 以 pass^k 度量可靠性?
- □ 可觀測性:trace 是否含 config 版本、token、工具錯誤、終態? 配置是否作為單一版本化單元?
- □ 成本: 是否有硬預算上限 (輪次/呼叫/token/時長)?

是否做 model/effort 分診? 成本進監控?

七、削減式審查 (第一、十四、二十三章)

- □ 每個 harness 組件是否標注了「它補償模型的哪個缺陷 X」?
- □ 哪些組件是地基 (注意力/安全/驗證約束, 持久)、哪些是鷹架 (模型癖性補償, 應標注壽命)?
- □ 上次模型升級後, 是否重估並刪除了已失效的補償組件?

全書終。