AI 時代的產品開發思維

不用會寫程式，也能用 AI 打造完整軟體產品的方法論

呂紹民（Darren Lu）

2026

前言
Part 0｜在開始之前
第 1 章｜AI 改變了什麼，沒改變什麼
　改變了的事
　沒改變的事
　關鍵翻轉：從 HOW 到 WHAT 和 WHY
　那我需要學什麼？
第 2 章｜指揮 AI 的前提：你腦中要有地圖
　軟體產品的分層架構
　蓋房子的比喻
　每一層的白話翻譯
　你不需要知道怎麼做，但要知道它們存在
　其他你該知道的概念
　這本書就是那張地圖
Part 1｜知識地圖 — 軟體產品的組成
第 3 章｜全貌：一個軟體產品長什麼樣
　架構全貌
　你不需要知道每一層怎麼做
　回到謝卡醬
第 4 章｜前端：使用者看到的一切
　前端做什麼
　前端技術的選擇
　四個檢查點
　前端的地雷
第 5 章｜後端：幕後的工作

什麼東西該放後端
TutorKit 的故事
從我的經驗，後端最該注意的四件事
第 6 章｜API：前後端的溝通語言
API 的四種動作
一個真實的 API 長什麼樣
API 設計時我會跟 AI 確認的事
API 是合約
第 7 章｜資料庫：產品的記憶
Excel vs 資料庫
資料庫決策的真實考量
TutorKit 和 ExamBank 的共用經驗
第 8 章｜第三方服務：不用自己造輪子
常見的第三方服務
判斷力：用現成的還是自己做
風險
第 9 章｜部署：從「我的電腦能跑」到「全世界都能用」
為什麼不能用自己的電腦
部署的流程
不同平台適合什麼
部署前我會確認的事
第 10 章｜維運：產品是活的
養店的日常
沒有 100% 不壞的系統
監控是什麼
一個半夜掛掉的故事
維運的心態
Part 2｜方法論 — 怎麼用 AI 走完全程
第 11 章｜核心方法：分層下指令
四層對話法
常見的指令錯誤
第 12 章｜讓 AI 自己研究，你負責決策
一個真實的範例
你的工作不是選「最好的」
問 AI 問題的技巧
第 13 章｜CLAUDE.md：教 AI 認識你的專案
什麼是 CLAUDE.md
該寫什麼

禁止事項比允許事項重要
第 14 章｜迭代思維：先求有，再求好
五輪迭代法
第 15 章｜驗收的能力比指揮的能力更重要
一次慘痛的教訓
三個黃金問句
驗收的重點是使用者角度
Part 3｜實戰流程 — 從想法到上線
第 16 章｜Phase 1：定義（30 分鐘）
你不需要想很完整才能開始
怎麼把想法變成結構化需求
AI 會幫你把模糊的想法變清楚
第 17 章｜Phase 2：建造（數小時到數天）
一次一個功能
怎麼決定功能的順序
驗收就是自己用用看
時間預期
第 18 章｜Phase 3：測試（1–2 小時）
餐廳的試菜
三層測試
你不需要自己寫測試
測試不是可選的，是保險
第 19 章｜Phase 4：上線（1 小時）
我的第一次部署
上線前的思考清單
環境變數：你的產品的秘密
部署平台怎麼選
上線後第一件事
第 20 章｜Phase 5：迭代（持續進行）
上線不是結束
根據真實回饋迭代
監控：知道你的產品還活著
永不結束的循環
Part 4｜心智模型 — 長期受用的思維方式
第 21 章｜懂得怎麼問，比什麼都懂更重要
問對問題比找對答案重要
正確問法的公式
AI 是隨身導師，但你要主動追問

即使有技術背景，情境不同就是新的挑戰
第 22 章｜分辨「AI 該決定的」vs「你該決定的」
技術決策：放手讓 AI 來
產品決策：你自己來
三類決策對照表
一個簡單的判斷法
第 23 章｜安全意識與成本意識
安全：五件刻在腦子裡的事
成本：你以為免費，其實有上限
你不需要變成資安專家或財務長
附錄 A｜知識地圖速查表
附錄 B｜跟 AI 對話的黃金問句
一個額外的提醒
附錄 C｜常見的「我不知道自己不知道」
後記

前言

嗨，我是紹民。

資工系畢業，做了六年的產品經理，現在在台灣一家網路認證公司負責產品創新。我會寫程式，工作上也常跟工程師討論技術細節，但我的主要工作一直是搞清楚「該做什麼」而不是「怎麼寫」。

先講為什麼會有這本書。

這兩年 AI 工具爆發，我身邊越來越多人想用 AI 來做自己的產品。有些是想創業的朋友，有些是公司裡想推動新東西的 PM 或設計師，還有一些本身就是工程師但沒做過完整產品的人。他們卡住的地方，通常不是「不會寫程式」這件事，說真的，現在 AI 可以幫你寫大部分的程式碼。真正卡住的是：他們不知道一個軟體產品的全貌長什麼樣子。

什麼是前端、什麼是後端？資料庫要怎麼選？部署是什麼意思？網域怎麼設定？金流串接要注意什麼？SEO（Search Engine Optimization，讓搜尋引擎更容易找到你的網站）是在產品的哪個階段才要管的？這些東西分開來看都不難，但沒有人幫你把它們串在一起看過，你就是會覺得眼前一片迷霧。

所以這本書想做的事很單純：給你一張知識地圖。

我不會教你寫程式。市面上教寫程式的資源已經多到看不完了，何況現在有 AI 幫忙，寫程式本身的門檻已經低很多。這本書要幫你建立的是對整個軟體產品的理解，從概念到上線，中間會經過哪些環節、每個環節在做什麼、你需要做什麼決定。

有了這張地圖，你在用 AI 做產品的時候就不會迷路。你會知道自己現在在哪裡、下一步該往哪走、遇到問題該搜尋什麼關鍵字。

書裡會穿插不少我自己做 Side Project 的真實經驗。過去這段時間我做了十六個專案，從婚禮邀請函、考題庫平台，到有在收費的 SaaS（Software as a Service，透過網路提供的軟體服務）都有。有些做得還不錯，有些踩了很多坑，半夜被 API 搞到想摔電腦的時刻也不是沒有。這些經驗我會盡量誠實地寫出來，因為我覺得知道「別人在哪裡跌倒」比看一堆理論有用得多。

最後講一下這本書怎麼讀。你當然可以從第一章開始按順序讀，整本書的結構就是按照做一個產品的流程來安排的。但如果你已經有一些基礎，或者你現在正好卡在某個特定的環節，直接跳到你需要的章節也完全沒問題。每個章節都是相對獨立的，我會在需要的地方標出跟其他章節的關聯。

好，那就開始吧。

紹民 2026 年，台北

Part 0｜在開始之前

第 1 章｜AI 改變了什麼，沒改變什麼

我是一個資工系畢業、做了六年的產品經理。我知道程式怎麼寫，但我的日常工作是定義產品、管理專案，不是寫程式碼。

兩年前如果有人問我會不會自己做一個完整的軟體產品，我會說：技術上我都懂，但時間不允許。光前端就得搞半天，後端再搞半天，部署又是另一回事。一個人做這些，投資報酬率太低。

現在我可以接著說：「不過我用 AI 做了十六個 Side Project。」

在過去這段時間，我做了婚禮邀請函網站、考題庫平台、AI 口述歷史工具、桌面聽寫 App、自動化影片管線、婚禮感謝卡 SaaS。這些不是玩具，有些真的有人在用，有些有在收費。

一個產品經理，不靠工程團隊，自己把東西從零做到上線。這件事到底是怎麼發生的？

改變了的事

最明顯的改變是寫程式的門檻降低了很多。

以前你想做一個網站，得自己寫 HTML、CSS、JavaScript，學一個前端框架，學後端語言，學資料庫，學部署。我知道每一個環節要做什麼，但實際動手全部串起來，工作量就是一整個工程團隊的份。

現在呢？我打開終端機，用自然語言跟 AI 說：「幫我建一個婚禮邀請函網站，要有倒數計時、Google Maps 地圖、RSVP（法文縮寫，意思是「請回覆」）表單。」它就開始生成程式碼了。

我第一次這樣做的時候，坐在電腦前看著程式碼一行一行冒出來，心裡的感覺不是興奮，比較像是在重新理解「做產品」這件事的邊界。過去我腦中規劃好的東西，要排進工程師的 Sprint（工程團隊的開發週期，通常兩週一輪）裡等兩週。現在我自己花一個晚上就能做出可以用的版本。

第二個改變是：一個人可以當一個團隊。

傳統的軟體開發，你至少需要前端工程師、後端工程師、設計師，可能還要一個 DevOps（負責部署和維運的工程角色）。一個最小可行的團隊大概要三到五個人。現在一個人配上 AI，可以同時處理前端介面、後端邏輯、資料庫設計、伺服器部署。

我做那十六個專案的時候，沒有任何一個有其他工程師參與。全部都是我一個人加上 AI。

這聽起來很厲害對吧？先別急，因為接下來要講的才是重點。

沒改變的事

AI 沒有改變的第一件事：你還是得知道軟體是由哪些部分組成的。

我做婚禮邀請函網站的時候，一開始很順利。AI 幫我生成了漂亮的頁面，有倒數計時、有地圖、有動畫效果。

然後我想加一個 RSVP 功能，讓賓客可以線上回覆出不出席。AI 也幫我寫好了表單，但我按下送出之後，什麼都沒發生。

因為表單需要一個後端伺服器來接收資料，還需要一個資料庫來存資料。而我只有前端頁面。

身為 PM，我當然知道前後端分離這回事。但當你自己一個人動手做，不再有工程師幫你把架構搭好的時候，這種「知道」跟「親手處理」之間的落差就會跑出來。你其實懂，但你得主動把腦中的架構知識轉化成對 AI 的具體指令。

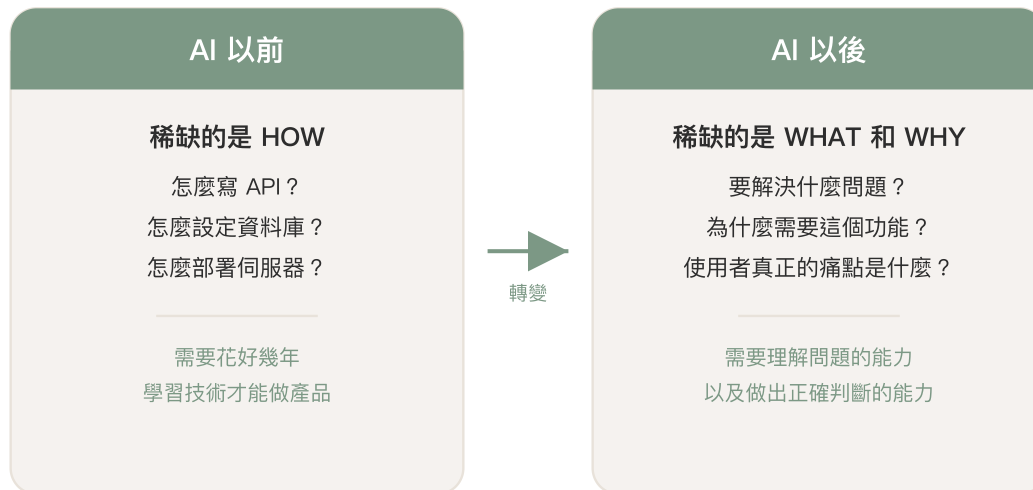
AI 沒有改變的第二件事：你還是得能判斷 AI 做得對不對。

AI 不是神，會犯錯，而且犯的錯經常看起來很有道理。很有自信地告訴你一段程式碼可以解決你的問題，然後那段程式碼跑下去直接把你的資料庫清空。

如果你不具備判斷能力，你就沒辦法做品質把關。你會變成一個只能按「確認」的橡皮圖章，而不是一個真正在指揮開發的人。

關鍵翻轉：從 HOW 到 WHAT 和 WHY

我們來看看新舊時代的核心差異。



圖：從 HOW 到 WHAT/WHY 的轉變

以前的世界，技術能力是最值錢的東西。你會寫程式，你就是稀缺人才。公司花大錢請資深工程師，因為他們知道 HOW，知道怎麼把東西做出來。

現在 AI 可以搞定大部分的 HOW。寫程式的速度比人快十倍，而且不用休息、不會請假、不會因為需求改了第七次就暴怒。

但 AI 不知道 WHAT，你到底要做什麼。也不知道 WHY，為什麼要做這個而不是做那個。

這些判斷，是人的工作。是你的工作。

我做婚禮邀請函的時候，AI 不會告訴我：「你應該把 RSVP 的截止日期設在婚禮前兩週，因為你需要提前跟餐廳確認桌數。」這是我自己知道的事情，因為我正在籌備自己的婚禮。

AI 也不會告訴我：「你的邀請函不需要做多國語言，因為你的賓客都是台灣人。」這種產品判斷，靠的是對使用者的理解。

在 AI 時代，知道要做什麼和為什麼要做，比知道怎麼做更重要。而這個能力，恰好是產品經理每天在練的事。

那我需要學什麼？

你不需要變成工程師。你需要的是一張地圖。

蓋房子不需要會砌磚，但你得知道房子有地基、有牆壁、有屋頂、有水電管線。你不需要自己去挖地基，但你得知道沒有地基的房子會倒。

軟體也一樣。你不需要會寫每一行程式碼，但你得知道一個軟體產品是由哪些部分組成的，它們之間怎麼互動，哪些地方容易出問題。

這就是這本書要做的事。

下一章，我們來看那張地圖。

第 2 章｜指揮 AI 的前提：你腦中要有地圖

我在做 ExamBank（考題庫平台）的時候，被一個叫做 CORS 的東西卡了整整一天。

那天的狀況是這樣的：前端網站做好了，後端 API 也做好了，兩邊分開測試都沒問題。但是當我在瀏覽器上打開網站，讓它去呼叫後端 API 的時候，瀏覽器噴了一個紅色的錯誤訊息，大概是說 CORS policy 擋住了請求。

我知道 CORS（Cross-Origin Resource Sharing，瀏覽器的跨網域安全機制）是什麼。但第一次自己一個人處理前後端分離的部署時，還是被這個細節卡了。以前在公司，這種事情工程師在部署的時候就順手處理掉了，我根本不需要碰。

CORS 本身不難，真正的重點是：如果你腦中有整體架構的認知，就會知道這種問題該往哪個方向排查。

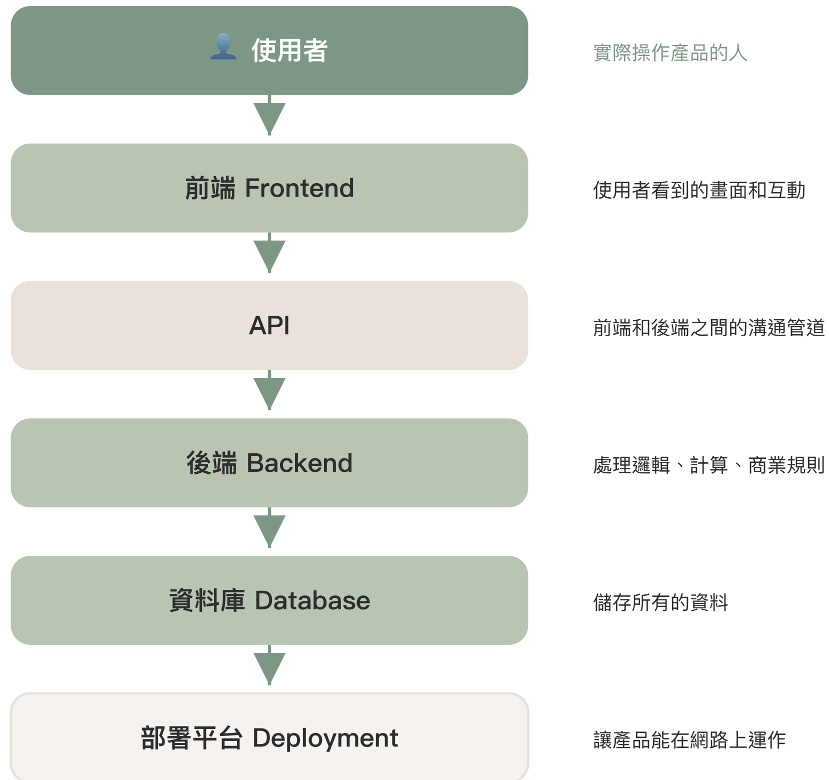
我應該在一開始就問 AI：「我的前端和後端是不同網域的，部署時有什麼要注意的嗎？」然後 AI 會告訴我要設定 CORS，五分鐘搞定。

但那天我沒有從架構的角度去思考，而是直接盯著錯誤訊息 debug，繞了一大圈。這就是「知道概念」跟「實戰時想得起來」之間的差距。

這就是為什麼你需要那張地圖，不只是存在腦中，還要隨時能調出來用。

軟體產品的分層架構

一個現代的軟體產品，大致上分成四層。



圖：軟體產品分層架構

最上層是使用者直接互動的前端（Frontend），就是你看到的按鈕、文字、圖片、表單。技術上常用 HTML、CSS、JavaScript，搭配 React、Next.js 這些框架。

前端下面是後端（Backend），處理邏輯的地方。驗證身分、計算價格、檢查庫存，這些看不到但讓一切運作的事情都在後端發生。常用 Python、Node.js、FastAPI、Express。

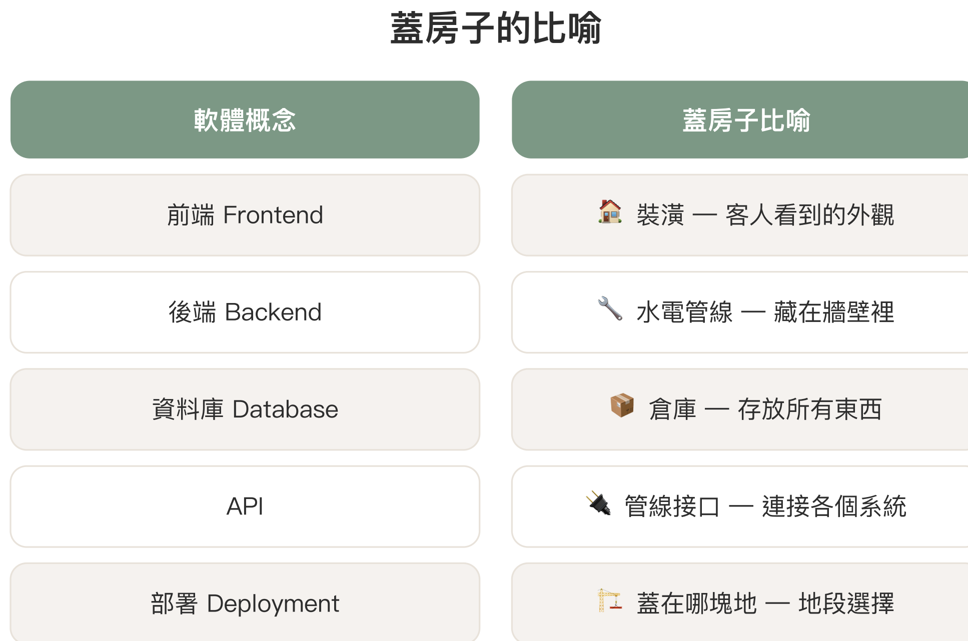
後端再往下是資料庫（Database），存放所有資料的地方。會員帳號、訂單紀錄、上傳的檔案路徑，全部在這裡。常見的有 PostgreSQL、MySQL、Supabase。

最後是部署平台（Hosting / Deployment），把你的軟體放到網路上讓別人用。常見的有 Cloudflare、Google Cloud、Vercel。

這四層之間透過 API（Application Programming Interface）溝通。前端不能直接去翻資料庫，它得透過 API 這個管道來跟後端要資料。我被 CORS 卡住的那次，問題就出在這個管道上。

蓋房子的比喻

用蓋房子來對照會更直覺。



圖：軟體架構與蓋房子的對照

前端就像室內裝潢和門窗，看得到、摸得到的部分。後端就像水電管線和結構工程，看不到但讓一切運作。資料庫就像倉庫，存放所有東西的地方。API 就像管線接口，讓資料在不同系統之間流通。部署就像蓋在哪塊地上，地段決定了租金和可及性。

你不需要會接水管，但你要知道房子裡有水管這件事。不然等到廁所不通的時候，你可能會以為是地板壞了。

軟體開發也一樣。你不需要會寫 API，但你要知道前端和後端之間是透過 API 在溝通的。不然等到出問題的時候，你連問題出在哪一層都搞不清楚。

每一層的白話翻譯

前端（Frontend）是使用者直接互動的那一層。你在手機上看到的每一個按鈕、每一行文字、每一張圖片，都是前端的工作。它就像餐廳的用餐區，客人看到的、坐到的、吃到的都在這裡。

後端（Backend）是在幕後處理邏輯的那一層。當你按下「送出訂單」，後端會驗證你的身分、計算價格、檢查庫存。它就像餐廳的廚房，客人看不到，但所有真正的工作都在這裡發生。

資料庫（Database）是存資料的地方。你的帳號密碼、你的訂單紀錄、你上傳的照片路徑，全部存在資料庫裡。它就像餐廳的食材倉庫，廚房需要什麼食材，就去倉庫拿。

API（Application Programming Interface）是前端跟後端之間的溝通管道。前端不能直接衝進後端翻資料，它得透過 API 這個「服務窗口」來點餐和取餐。

部署（Deployment）是把你的軟體放到網路上的過程。你在自己電腦上做好的東西，別人是看不到的。你得把它「部署」到某個伺服器上，就像你在家做好了一桌菜，得端到餐廳去才有客人吃得到。

你不需要知道怎麼做，但要知道它們存在

我做 ExamBank 的時候，整個架構是前端部署在 Cloudflare Pages，後端跑在 Google Cloud Run，資料庫用 Supabase。

圖：ExamBank 三層架構

我不需要知道 Cloudflare Pages 底層是怎麼運作的。我不需要知道 Google Cloud Run 的容器技術細節。我不需要知道 Supabase 的 PostgreSQL 是怎麼做索引優化的。

但我需要知道：我的產品有三層，前端在 Cloudflare、後端在 Google Cloud、資料庫在 Supabase。它們是分開的，它們之間透過 API 溝通，而且這個溝通有一些規則（比如 CORS）。

有了這個認知，當出問題的時候，我至少能做到兩件事。

第一，我能縮小問題的範圍。「網站打不開」是前端的問題。「資料存不進去」可能是後端或資料庫的問題。「前端能開但資料顯示不出來」大概是 API 那一層的問題。

第二，我能問出正確的問題。與其跟 AI 說「我的網站壞了」（這等於跟醫生說「我不舒服」），我可以說「我的前端可以正常顯示，但呼叫後端 API 時收到錯誤回應，錯誤碼是 403」。這種精準的描述，會讓 AI 給你精準的答案。

這是 PM 的核心能力在 AI 時代的延伸：你不需要自己解問題，但你要能準確地定義問題。

其他你該知道的概念

除了那四層架構之外，還有幾個概念你遲早會碰到，我先簡單列在這裡。

版本控制（Git），追蹤程式碼修改紀錄的系統。就像 Google Docs 的版本紀錄，讓你可以回到之前的版本。你不需要精通 Git，但你要知道它存在，而且它是避免災難的救命繩。

環境變數（Environment Variables），存放密碼、API 金鑰等敏感資訊的地方。你不會把家裡的鑰匙直接貼在門上，同理你不該把密碼直接寫在程式碼裡。

第三方服務（Third-party Services），別人做好的功能，你直接拿來用。寄 email 用 Resend、處理付款用 PayUNI、做使用者登入用 Supabase Auth。不需要自己從頭造。

這些概念我們在後面的章節都會一一展開。現在你只需要知道它們存在，先把名字記住就好。

這本書就是那張地圖

回到最初的比喻。

你要指揮 AI 幫你蓋房子，你不需要自己會砌磚、接水管、裝電線。但你需要一張建築藍圖，上面標示了哪裡是地基、哪裡是承重牆、哪裡是水管、哪裡是電線。

沒有這張藍圖，你可能叫 AI 把承重牆拆掉來擴大客廳，結果整棟房子塌了。你可能不知道為什麼馬桶不通，因為你忘了有水管這東西。你可能蓋了一棟很漂亮的房子，但忘了留大門，因為太專注在室內裝潢。

這本書要給你的，就是那張軟體產品的建築藍圖。

接下來的每一章，我們會一層一層地看過去。不講程式碼，只講概念。你讀完之後不會變成工程師，但你會變成一個知道怎麼指揮 AI 的人。

而在這個時代，那可能比會寫程式更有用。

Part 1｜知識地圖 — 軟體產品的組成

第 3 章｜全貌：一個軟體產品長什麼樣

我真正理解「軟體產品」的結構，是在做謝卡醬（weddingcard-saas）的時候。

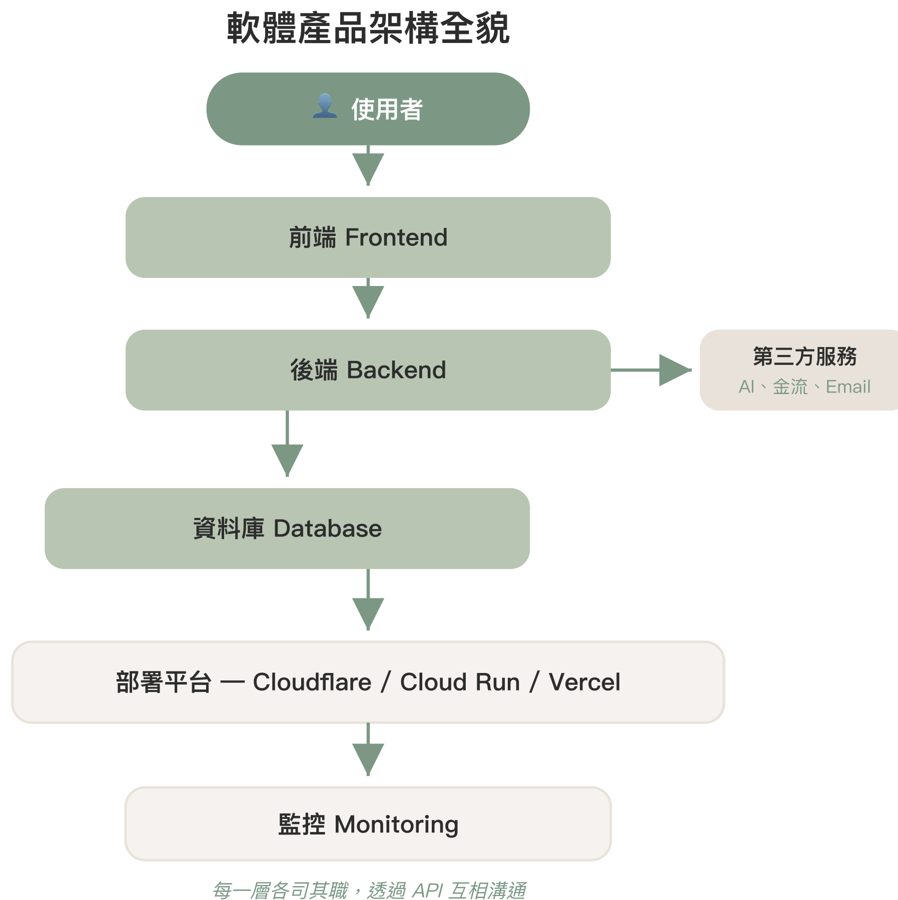
婚禮感謝卡的線上服務，新人上傳賓客名單，系統自動生成個人化的感謝小卡，用 Email 寄出去。聽起來功能單純。

但一列需求就不單純了：要有網頁讓新人操作、要有地方存賓客資料、要能生成卡片圖片、要串金流讓人付費、要能寄 Email、要部署到網路上讓所有人都能用。這些需求攤開來，才看到一個軟體產品真正的規模。

使用者看到的只是冰山一角，底下有一整座架構在運作。

架構全貌

一個典型的軟體產品，大概是這樣分層的：



圖：軟體產品架構全貌

每一層負責一件事，用一句話解釋：

- 前端：使用者看得到、摸得到的介面
- 後端：使用者看不到，但負責所有「真正的工作」
- API：前端跟後端之間的溝通橋樑
- 資料庫：所有資料的儲存倉庫
- 第三方服務：別人做好的功能，你付錢就能用
- 部署：把東西放上網路，讓全世界都能連
- 監控：上線之後持續盯著，確保沒出事

你不需要知道每一層怎麼做

這是本章最重要的觀念。

你不需要知道前端怎麼切版面，不需要知道後端怎麼寫邏輯，不需要知道 SQL 語法怎麼查資料。這些 AI 都會做。

但你需要知道這些層「存在」，而且知道它們各自負責什麼。因為當你跟 AI 協作的時候，你得能說出：「這個功能應該放在前端還是後端？」「我的資料存在哪裡？」「使用者付完錢之後，資料流是怎麼走的？」

不知道這些層存在的人，就像是走進一間餐廳但不知道有廚房，他會困惑為什麼菜不是服務生做的。

回到謝卡醬

讓我用謝卡醬的例子把每一層對應起來：

層	謝卡醬裡的樣子
前端	新人看到的管理後台、賓客看到的感謝卡頁面
後端	驗證賓客身份、生成卡片、處理付款回調
API	前端送出「查詢這位賓客」的請求，後端回傳賓客資料
資料庫	賓客名單、專案設定、付款紀錄
第三方服務	Resend 寄 Email、PayUNI 收款、Cloudflare R2 存圖片
部署	Cloudflare Workers
監控	錯誤日誌、API 回應時間

當我第一次把這張表畫出來，才真正搞懂自己到底在做什麼。在那之前，腦子裡就是一團「要做一個網站」的模糊概念。畫出來之後，每一層變成了一個可以單獨討論、單獨交給 AI 去處理的任務。

這就是知識地圖的價值：你不需要會開車，但你得看得懂地圖。

第 4 章｜前端：使用者看到的一切

前端就是裝潢。

想像你開了一家店，前端就是客人走進來看到的一切：牆壁的顏色、櫃檯的位置、菜單掛在哪裡、椅子舒不舒服。客人不會知道廚房長什麼樣、水電怎麼接的、排油煙機是哪個牌子，他們只在乎走進來的感覺。

前端就是那個感覺。

前端做什麼

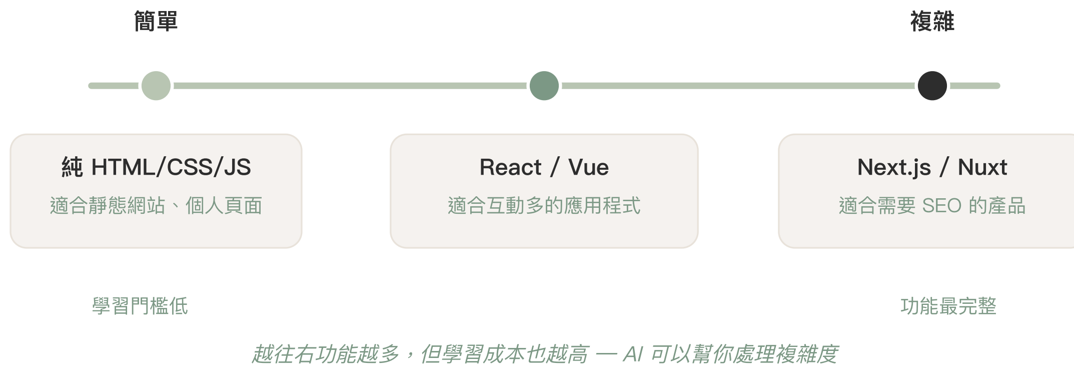
具體來說，前端負責這些事：

- 畫面上有哪些東西（文字、圖片、按鈕、表單）
- 這些東西怎麼排列（手機上長什麼樣、電腦上長什麼樣）
- 使用者點了按鈕會發生什麼（跳轉頁面、彈出視窗、送出表單）
- 動畫效果（淡入淡出、滑動、載入中的轉圈圈）

前端不負責的事：判斷密碼對不對、從資料庫撈資料、處理付款。這些是後端的工作，下一章會講。

前端技術的選擇

做前端有很多種方式。我用過的大概分三類：



圖：前端技術光譜

你完全不需要自己選。當你跟 AI 說「我要做一個品牌官網」，它會推薦適合的方案。但你可以知道這個光譜的存在。

我做皂花拾光（soap-flower-studio）的時候，就是用最簡單的純 HTML。一個 `index.html`、一個 `style.css`、一個 `main.js`，三個檔案搞定一整個品牌官網，有 Lightbox 相簿、有 FAQ、有 SEO、有 RWD（Responsive Web Design，讓網頁在不同螢幕大小都能正常顯示），部署到 Cloudflare 上，每月流量費用是零。

不是所有東西都需要用最新最潮的框架。有時候最樸素的方案反而最適合。

四個檢查點

你不需要會寫 CSS，不需要知道 `useState` 是什麼，不需要搞懂 Webpack 設定。但每次看到前端成品，記得跑一遍這四項：

1. 手機版看起來怎麼樣？現在超過一半的流量來自手機，你的網站在小螢幕上不能爛掉。
2. 搜尋引擎找得到嗎？這叫 SEO。如果 Google 搜不到你的網站，等於不存在。
3. 載入速度夠快嗎？使用者等超過三秒就會離開。圖片太大、程式碼太肥，都會拖慢速度。
4. 視障者能用嗎？這叫可及性（Accessibility）。按鈕有沒有文字描述、對比度夠不夠高，這些不只是道德問題，在某些國家是法律要求。

跑完這四項，AI 會幫你處理技術細節。

前端的地雷

有一個常見的誤會：很多人以為前端只是「把東西弄漂亮」。

不是。前端是使用者體驗的第一道防線。如果表單送出後沒有任何回饋，使用者會連按十次；如果錯誤訊息寫「Error 500」，使用者會直接關掉；如果頁面在手機上要左右滑才看得到內容，使用者根本不會看。

漂亮是加分，好用是基本。

第 5 章｜後端：幕後的工作

如果前端是裝潢，後端就是廚房。

客人坐在裝潢精美的餐廳裡，看著菜單點了一道牛排。他不會看到廚房裡面有人在切肉、有人在顧火候、有人在洗盤子、有人在核對食材庫存。他只會在適當的時間收到一盤牛排。

後端就是那個廚房。使用者看不到，但所有真正需要計算、判斷、儲存的工作，都在這裡發生。

什麼東西該放後端

有一個很簡單的判斷標準：讓使用者的瀏覽器自己做這件事，會有問題嗎？

如果答案是「會」，那就放後端。

功能	放哪裡	為什麼
顯示動畫效果	前端	純視覺，不涉及敏感資料
驗證使用者密碼	後端	密碼不能讓瀏覽器看到
呼叫 AI 解題	後端	API Key 不能暴露在前端
處理信用卡付款	後端	金流資料必須在安全環境處理
寄送 Email	後端	需要 SMTP（寄信用的通訊協定）憑證
計算購物車小計	前端	純計算，方便即時顯示
產生最終訂單金額	後端	防止前端被竄改價格

核心邏輯只有兩條：一、涉及安全性的東西不能放前端（API Key、密碼、金流）。二、使用者可能動手腳的東西不能只在前端驗證（價格、權限、數量）。

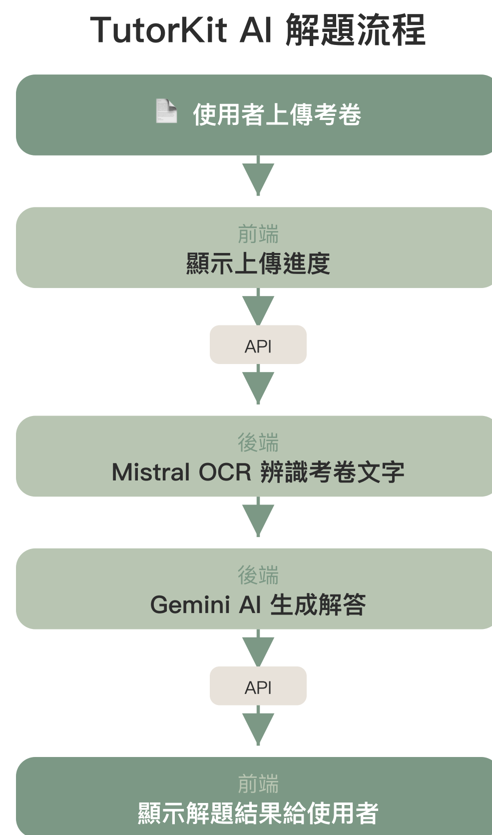
TutorKit 的故事

我做 TutorKit（智慧考卷解析）的時候，有一個功能是用 AI 解題，使用者上傳一張考卷照片，系統用 OCR（光學字元辨識，把圖片裡的文字轉成可編輯的文字）辨識題目，再用 Gemini AI 生成解答。

最初考慮過一個偷懶的做法：直接在前端呼叫 Gemini API。少寫一層後端，省事。

但問題很明顯：如果把 Gemini 的 API Key 放在前端的程式碼裡，任何人只要按 F12 打開瀏覽器的開發者工具，就能看到那把 Key。然後他就可以拿我的 Key 去呼叫 Gemini，帳單算我的。

所以 TutorKit 的 AI 解題流程是這樣的：



圖：TutorKit AI 解題流程

使用者從頭到尾只看到「上傳 → 等待 → 結果出來了」。他不知道中間經過了兩個 AI 服務，也不需要知道。但如果沒有後端，這整個流程就無法安全地運作。

從我的經驗，後端最該注意的四件事

跟前端一樣，你不需要會寫後端程式。但從我做十幾個專案的經驗，這四個問題一定會碰到：

「這個資料放前端還後端處理？」判斷標準就是上面說的，會有安全問題嗎？

「同時很多人用會怎樣？」這叫併發（Concurrency）。十個人同時上傳考卷，後端撐得住嗎？在 TutorKit 上線初期，我就遇過三個人同時上傳大檔案，後端 timeout 的狀況。

「如果這個步驟失敗了怎麼辦？」付款到一半網路斷了，資料會不會不一致？

「敏感資料怎麼保護？」使用者的個資、密碼、付款資訊，怎麼確保不會外洩？

這些問題你先想過一輪，再跟 AI 協作的時候，產出的品質會完全不同。

第 6 章｜API：前後端的溝通語言

API 就是菜單。

在餐廳裡，你不會走進廚房跟廚師說「我要一塊肉，用橄欖油煎，五分熟，旁邊放一些蘆筍」。你看菜單，選「嫩煎菲力，五分熟」，服務生把你的選擇傳到廚房，廚房按照規格做好再端出來。

API（Application Programming Interface）就是這份菜單。它定義了前端可以跟後端「點」什麼、怎麼點、會拿到什麼。

API 的四種動作

菜單上的動作其實很固定，就四種：

動作	餐廳比喻	軟體裡的意思
GET	「我要看菜單」	讀取資料
POST	「我要點這道菜」	新增資料
PUT	「我改成七分熟」	修改資料
DELETE	「取消那道菜」	刪除資料

就這四種。世界上絕大多數的軟體功能，都可以用這四種動作組合出來。

一個真實的 API 長什麼樣

我做 ExamBank（考題庫平台）的時候，前端需要跟後端溝通的事情很多：顯示考題列表、搜尋特定科目、上傳新考題、修改考題內容。

這些需求變成 API 的樣子大概像這樣：

```
GET    /api/exams              → 取得考題列表
GET    /api/exams/123          → 取得第 123 號考題的詳細內容
POST   /api/exams              → 新增一份考題
PUT    /api/exams/123          → 修改第 123 號考題
```

```
DELETE /api/exams/123          → 刪除第 123 號考題

GET    /api/subjects            → 取得所有科目列表
GET    /api/schools             → 取得所有學校列表
```

看起來很規律對吧？確實。好的 API 設計就是這樣，一看就知道這個端點（endpoint）是做什麼的。

你不需要會寫這些 API，AI 可以幫你生成。但你需要能看懂這張清單，確認「對，我的產品需要這些功能」。

API 設計時我會跟 AI 確認的事

以謝卡醬為例，當 AI 幫我設計驗證賓客的 API 時，我的確認清單是這樣的：

- 這支 API 需不需要登入才能呼叫？（賓客查詢不用，但管理後台的 API 要）
- 回傳的資料會不會太多？（我只需要賓客姓名和桌次，不需要連電話都回傳給前端）
- 找不到這位賓客的時候，前端會收到什麼？（要有清楚的錯誤訊息，不能只丟一個 404）
- 有人惡意一直打這支 API 怎麼辦？（我設了 Rate Limiting，每分鐘每個 IP 只能打 5 次，防止暴力猜測賓客資料）

每個產品的確認清單不一樣，但邏輯是一樣的：安全、效率、錯誤處理。

API 是合約

最後一個觀念：API 是前端和後端之間的合約。

一旦雙方同意「用 GET /api/exams 可以拿到考題列表，回傳格式長這樣」，前端就按這個格式來解析，後端就按這個格式來回傳。改了任何一邊沒通知另一邊，東西就會壞掉。

這就是為什麼在開發過程中，API 的設計通常是最先確定的事情之一。先把菜單定下來，廚房跟外場才能各做各的。

第 7 章｜資料庫：產品的記憶

資料庫就是一個有規則的 Excel。

我知道這個比喻聽起來太簡化了，但對於理解「資料庫是什麼」來說，它真的很好用。

Excel vs 資料庫

假設你用 Excel 管理婚禮賓客名單，有姓名、電話、桌次、是否回覆這幾個欄位。資料庫裡面長得幾乎一模一樣，差別在欄位名稱會用英文（name, phone, table_no, rsvp），而且每筆資料會有一個唯一的 id。

那為什麼不直接用 Excel？因為 Excel 做不到這些事：

能力	Excel	資料庫
100 人同時編輯同一份資料	會打架	正常運作
規定「電話欄位不能空白」	要靠人自律	系統強制
「刪除某位賓客時，他的賀卡也跟著刪」	要手動處理	自動連動
資料量超過十萬筆	會卡	不會
網站掛了重開，資料還在嗎	看你有沒有存	永遠都在
限制「只有管理員能看電話號碼」	做不到	可以設定

資料庫就是一個被軟體工程師強化過的 Excel，多了併發控制、權限管理、資料驗證、自動備份這些能力。

資料庫決策的真實考量

在做 wedding-guest-cards（婚禮賀卡系統）的時候，我需要回答一個問題：資料要存在哪裡？這個系統的賓客驗證需要很快的回應速度，因為賓客在婚禮現場掃 QR Code 的時候，等兩秒鐘都嫌久。所以除了 Supabase 之外，我還用了 Redis 做快取。

這些選擇都是從使用情境回推的。你的使用者在什麼場景下用你的產品？那個場景對速度的要求是什麼？答案會直接影響資料庫的選擇。

TutorKit 和 ExamBank 的共用經驗

這裡分享一個真實的決策。我做了兩個教育相關的產品：TutorKit（智慧考卷解析）和 ExamBank（考題庫平台）。它們都需要資料庫，而且都用 Supabase（一個雲端資料庫服務）。

問題來了：要開兩個 Supabase 帳號，還是共用一個？

我選了共用。原因很實際，Supabase 的免費方案有額度限制，開兩個帳號就是兩倍費用。但共用也有代價：兩個產品的資料表會混在一起，所以我把 ExamBank 的資料表全部加上 `exambank_` 前綴來區分。

這不是什麼高深的架構決策，就是預算有限下的務實選擇。但它說明了一件事：資料庫的選擇不只是技術問題，也是成本問題。你要能跟 AI 說：「我這個專案的預算是零，有沒有免費的方案？」

第 8 章｜第三方服務：不用自己造輪子

現代軟體有個特性：大部分功能不是自己寫的。

這不是偷懶，是合理的資源分配。你要做一個會員登入功能，與其自己寫一套密碼加密、Session 管理（就是「記住使用者已經登入」的機制，讓你不用每點一個頁面就重新輸入帳密）、忘記密碼流程，不如用 Google 或 Facebook 登入，他們的資安團隊比你多幾千人，處理這些問題的時間比你的產品存在的時間還長。

常見的第三方服務

這張表是我做了十幾個專案後整理出來的，幾乎每個產品都會用到其中幾項：

需求	常見服務	我用過的
使用者登入	Google OAuth, Auth0, Supabase Auth	Firebase Auth, Supabase Auth
收款	Stripe, PayPal, 綠界, PayUNI	PayUNI
寄 Email	SendGrid, Resend, AWS SES	Resend
AI 功能	OpenAI, Gemini, Claude, Mistral	Gemini, Claude, Mistral OCR
檔案儲存	AWS S3, Cloudflare R2, Supabase Storage	Cloudflare R2, Supabase Storage
資料庫	Supabase, Firebase, PlanetScale	Supabase
語音合成	Google TTS, Edge-TTS, ElevenLabs	Edge-TTS
地圖	Google Maps, Mapbox, Leaflet	Leaflet（開源免費）

判斷力：用現成的還是自己做

不是所有東西都該用別人的。判斷的原則是：這個功能是你產品的核心價值嗎？

如果是，自己做（或至少深度客製）。如果不是，用現成的。

謝卡醬的核心價值是「生成漂亮的個人化感謝卡」，這個我自己做。但「寄 Email」不是核心價值，我用 Resend；「收款」不是核心價值，我用 PayUNI。我不需要理解 SMTP 協議的細節，也不需要搞懂信用卡驗證的流程。

反過來，如果你做的是一個 Email 行銷平台，寄信就是你的核心價值，你大概不會想完全依賴 Resend。

風險

用第三方服務不是沒有代價。

漲價。你的產品剛上線時用的是免費方案，使用者變多之後超過免費額度，突然要開始付錢。Resend 的免費額度是每月 3,000 封 Email、每天 100 封。聽起來很夠？如果你的婚禮感謝卡服務一場婚禮就有 300 位賓客，三場就用完了。

倒閉。你依賴的服務如果收掉了，你的功能就跟著消失。這幾年 Heroku 砍免費方案、Firebase 調整定價，都讓很多小專案被迫搬家。

免費方案限制。很多服務的免費方案有功能限制、速度限制、或是會在你的產品上顯示他們的 Logo。

應對方式不複雜：做選擇的時候多看一下替代方案，不要把所有東西都綁在同一家服務上。如果某個服務真的掛了，你至少要知道可以換成什麼。

第 9 章｜部署：從「我的電腦能跑」到「全世界都能用」

「在我的電腦上可以用啊。」

這大概是軟體開發裡最經典的一句話。你在自己的筆電上做好了一個網站，打開瀏覽器看起來沒問題。然後你把網址傳給朋友，朋友說打不開。

當然打不開。你的筆電不是伺服器。

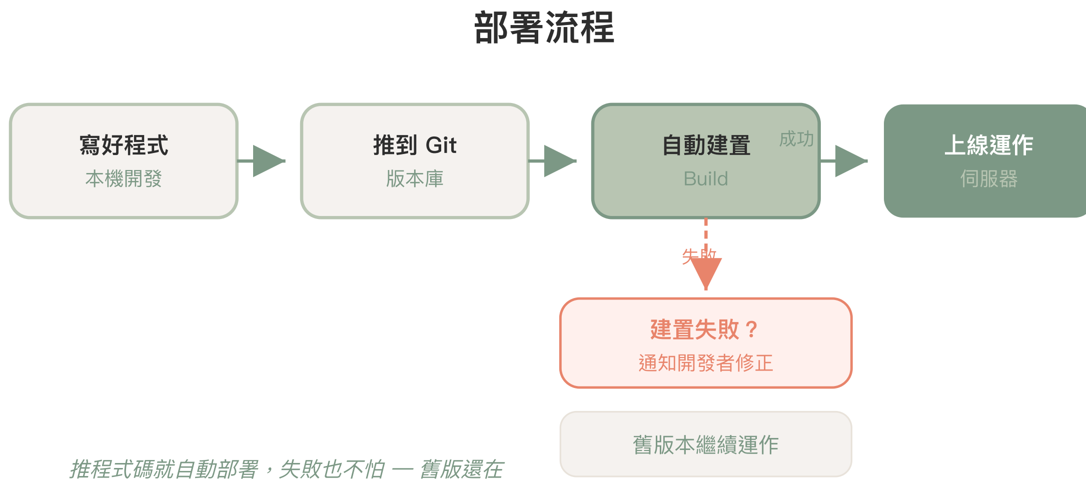
為什麼不能用自己的電腦

三個原因：

1. 你會關機。你睡覺的時候筆電蓋起來，你的網站就下線了。全年不關機，你的電費帳單會很精彩。
2. 你的網路太慢。家用網路的上傳頻寬通常很小，十個人同時連你的網站就卡住了。
3. 安全性。你的筆電直接暴露在網路上，等於幫駭客開了一扇門。

所以我們需要「部署」，把做好的東西放到專業的伺服器上，讓它 24 小時運作、有足夠的頻寬、有安全防護。

部署的流程



圖：CI/CD 部署流程

這個流程叫 CI/CD（持續整合與持續部署，推程式碼後自動測試和上線的機制），白話講就是「推了新程式碼就自動部署」。你不需要每次都手動把檔案上傳到伺服器，推一下 Git 就搞定。建置失敗的話，系統會通知你，舊版本繼續跑，不會影響線上使用者。

不同平台適合什麼

我做了十幾個專案，部署到過五個不同的平台。每個平台有它適合的場景：

平台	適合什麼	我的專案
Cloudflare Pages	靜態網站、前端	Travel_blog, ToolBox, ExamBank 前端, ohara-lab
Cloudflare Workers	輕量後端、邊緣運算	weddingcard-saas, soap-flower-studio
Google Cloud Run	完整後端、容器化、需要較多運算資源	TutorKit, ExamBank 後端, GuideCard, MemoryTale, CardButler
Vercel	Next.js 專案	OurStory

平台	適合什麼	我的專案
GitHub Pages	純靜態、零成本	Wedding（婚禮邀請函）

你不需要記住這些平台的差異。你需要知道的是：不同類型的東西適合放在不同的地方，而且這些平台很多都有免費方案。

部署前我會確認的事

每次要把新專案部署上線，我會跑一輪確認：

出事了能回到上一版嗎？這叫回滾（Rollback）。新版本有 bug，能不能一鍵退回上一個正常的版本？

環境變數怎麼管理？API Key、資料庫密碼這些不能寫在程式碼裡，要放在部署平台的環境變數設定中。

費用怎麼算？有些平台按流量收費，有些按運算時間收費。你的產品如果突然爆紅，帳單會不會跟著爆炸？

流量大了能自動擴展嗎？平常一天一百人用，某天上了新聞突然一萬人湧進來，撐得住嗎？

費用這點我有實際教訓。在 Google Cloud Run 上，幾個後端服務加上 Container Registry 的儲存費，每個月零零總總加起來比預期的高不少。免費方案的額度用完之後，那些看起來很小的數字會慢慢累積。所以費用一定要提早搞清楚。

第 10 章｜維運：產品是活的

很多人以為軟體做完就完了。

不是。軟體更像是開一家店，不是寫一本書。書印出來就定稿了，但店開了之後你得每天看營業狀況、處理客訴、更新菜單、修漏水的天花板。

養店的日常

軟體上線之後，你的日常大概長這樣：

每天要看的事： – 有沒有錯誤發生（Error Logs） – 系統有沒有正常回應（Health Check） – 有沒有異常流量（可能是有人在攻擊你）

每週可能要做的事： – 更新套件版本（修安全漏洞） – 回應使用者回報的問題 – 看一下費用有沒有超標

不定期要做的事： – 加新功能 – 效能優化（某個頁面突然變很慢） – 處理第三方服務的異動（API 改版、漲價、停止服務）

沒有 100% 不壞的系統

這個觀念很重要，但跟直覺相反：你的系統一定會壞。

不是「可能會壞」，是「一定會壞」。只是時間和方式的問題。資料庫連不上、第三方 API 掛了、某段程式碼有個 bug 只在特定條件下才會觸發、SSL（讓網站連線加密的安全機制）憑證過期了沒人注意到。

重點不是「怎麼讓系統永遠不壞」——那是不可能的。重點是壞了多快能發現、多快能修好。

這就是為什麼監控不是可選的。

監控是什麼

監控就是幫你的產品裝監視器。你不可能 24 小時盯著螢幕看系統有沒有出事，但你可以設定：

- API 回應時間超過 3 秒 → 通知你
- 錯誤率超過 1% → 通知你

- 月費用超過設定的上限 → 通知你
- 某個服務完全沒反應 → 通知你

通知的方式可以是 Email、Slack 訊息、甚至簡訊。重點是你在睡覺的時候如果系統出事了，你會被叫醒。

嗯，對，這不太浪漫。

一個半夜掛掉的故事

有一次我一個部署在 Cloud Run 上的服務，半夜突然開始回傳 502 錯誤。

原因是 Container 啟動失敗。Container 你可以想像成一個「打包好的便當盒」，你把程式碼和它需要的所有東西（套件、設定、執行環境）打包成一個盒子，丟到伺服器上就能跑。啟動失敗就像是便當盒打開發現裡面的食材壞了，整個沒辦法吃。

具體來說，前一天更新了一個套件的版本，新版本跟另一個套件不相容。這就像你換了一個新的食材供應商，結果他送來的醬油跟你原本的食譜搭不起來，整道菜做不出來。在自己電腦上測試的時候沒發現，因為本機的環境跟 Cloud Run 的環境有微妙的差異，就像你家的廚房能做出來的菜，換到另一個廚房，爐火大小不同、鍋具不同，結果就不一樣了。

發現的過程也很經典：早上起來看到使用者在 LINE 上跟我說「網站打不開」。我打開錯誤日誌，看到凌晨三點開始就全是 502。也就是說，我的服務掛了大約五個小時，我完全不知道。

從那之後我學到兩件事：第一，上線前要在跟正式環境一樣的環境測試。第二，一定要設監控通知。

我本來就知道該設監控。就是拖延。拖延的代價就是五個小時的服務中斷。

維運的心態

做產品跟寫程式是不一樣的心態。寫程式的時候你在「創造」，維運的時候你在「守護」。守護沒有創造那麼刺激，但如果你不守護，你創造的東西會慢慢壞掉。

套件不更新，安全漏洞就會累積。錯誤不處理，使用者就會流失。費用不監控，某天帳單會給你一個驚喜。

好消息是，AI 時代的維運比以前輕鬆很多。很多監控工具有免費方案，套件更新可以用自動化工具（Dependabot 之類的），錯誤排查可以把 Log 丟給 AI 幫你分析。

但「要去做」這個意識，得你自己有。AI 不會在半夜叫你起床看 Log——除非你先設定好了讓它叫你。

Part 2｜方法論 — 怎麼用 AI 走完全程

第 11 章｜核心方法：分層下指令

用 AI 寫程式最難的部分不是技術，是你不知道自己要什麼。

你想想：去餐廳點菜，「我要一碗牛肉麵」跟「我要吃東西」，廚師收到的指令品質差多少？AI 也一樣。它不怕你給的需求有技術含量，它怕你的指令模糊到它只能猜。而 AI 猜出來的東西，有時候看起來很像那麼回事，但其實完全不是你要的。

我在做 MemoryTale（一個 AI 口述歷史平台）的時候，一開始跟 AI 說：「幫我做一個可以記錄老人家故事的網站。」它做出來了，但跟我想的完全不一樣，它做了一個部落格。技術上沒錯，但那不是我要的東西。

後來我慢慢摸索出一套方法，我叫它「四層對話法」。不是什麼高深理論，就是踩坑踩出來的。

四層對話法

圖：四層對話法架構

第一層：目標

這一層你要說清楚「為什麼」跟「要什麼」，但不要說「怎麼做」。

回到 MemoryTale 的例子。我後來重新跟 AI 說的是：「我想做一個平台，讓年輕人可以用語音訪談的方式，幫家裡的長輩把人生故事記錄下來。訪談結束後，AI 會把錄音整理成一篇有結構的故事文章。」

注意差別。我沒說要用什麼技術，沒說資料庫要怎麼設計，我只說了：誰要用、用來幹嘛、最後產出什麼。這就夠了。AI 拿到這些資訊，就能幫你規劃整個架構。

第二層：決策

AI 規劃完架構後，會丟一堆技術選項給你。這時候你要用 PM 的思維來評估，問對的問題。

比如 AI 跟我說：「語音轉文字可以用 Google Speech-to-Text、Whisper、或 Deepgram。」我的做法是問：「這三個方案，哪個最便宜？哪個中文辨識最準？哪個最容易整合？幫我列一個比較表。」

AI 列完比較表，我就能用「我的情境」來做決定。重點是選最適合我的。這個概念後面幾章會反覆出現。

第三層：驗收

東西做出來了，你要檢查。重點不是逐行看程式碼，即使你看得懂，驗收的核心還是從使用者角度出發。打開網頁，點每個按鈕，走一遍流程。

「錄音功能正常嗎？」「轉出來的文字對嗎？」「故事文章的格式好讀嗎？」

這些才是真正重要的判斷。技術實作可以交給 code review 工具，但產品體驗只有人能把關。

第四層：迭代

驗收完發現問題，回頭跟 AI 說：「錄音超過十分鐘會斷掉，這要修。另外，故事文章的開頭太平淡了，能不能讓 AI 自動生成一個比較有吸引力的第一段？」

然後又回到第一層：新的目標、新的決策、新的驗收。循環下去。

常見的指令錯誤

我看過太多人在第一層就翻車了。這裡列三種典型的問法：

太模糊：「幫我做一個 App。」AI 會做出一個東西，但大概不是你要的。這就像跟裝潢師傅說「幫我弄一下房子」，他能幫你做，但做出來你肯定不滿意。

太具體：「幫我用 React 寫一個元件，用 useState 管理表單狀態，用 Axios 打 API，回傳的資料用 map 渲染成列表。」如果你會寫這種指令，你根本不需要 AI 幫你寫程式。而且，你把所有決策都做完了，但你的決策不一定是最好的。

剛好的層次：「我要做一個頁面，讓使用者填寫姓名和 Email 後送出，送出後顯示一個確認畫面。」清楚說了要什麼，沒有限定怎麼做。AI 有足夠的資訊知道你要什麼，又有足夠的空間用它覺得最好的方式來實現。

中間那個層次最難拿捏，但也最重要。說太少 AI 會亂猜，說太多你反而限制了它。我的經驗是：把自己當成一個在描述需求的 PM，而不是在寫技術規格書。

你可能會問：「那我怎麼知道自己的指令是不是在對的層次？」有個很簡單的判斷方法：如果你的指令裡出現了技術名詞，而且那些名詞其實跟你要解決的問題無關，那你大概說太具體了。退一步，用白話文重新講一次你要什麼。

第 12 章｜讓 AI 自己研究，你負責決策

做 Side Project 之前，我常常有一種焦慮：覺得自己要先把所有技術方案都研究過一遍，才能開始動手。

這個想法害我浪費了非常多時間。

後來我發現，AI 最擅長的事情之一就是幫你做功課。你不需要自己去 Google、看文件、比較方案。你只需要問對問題，然後從 AI 給你的選項裡做決定。

這才是正確的分工：AI 負責「研究」，你負責「判斷」。研究是可以被自動化的，判斷不行。因為判斷需要你對自己的情境有理解，而這個只有你自己知道。

一個真實的範例

做 EquityHunter（自動化股票研究引擎）的時候，我需要讓程式去呼叫 AI 來分析股票資料。問題是，市面上有好幾家 AI 可以選：Claude 的 API、Google 的 Gemini API、OpenAI 的 API⋯⋯我根本不知道該選哪個。

以前的我會花三天看文件、查價格、找別人的評測文章。現在的我直接問 AI：

「我要做一個股票研究引擎，需要 AI 分析公司財報資料然後產出研究報告。我的預算不高，每個月大概用不到一千次。幫我比較 Claude API、Gemini API、OpenAI API，從價格、分析品質、使用難度三個面向來看。」

AI 很快就列出了一個比較表。價格方面，Gemini 有免費額度，Claude 和 OpenAI 差不多。分析品質方面，Claude 在長文推理上評價不錯，但 Gemini 也不差。使用難度方面，三家都差不多。

你的工作不是選「最好的」

拿到比較表之後，我做了一個看起來很不「技術」的決定：先用 Claude API，因為我自己每天都在用 Claude Code，對它的風格比較熟悉。如果效果不好，再換 Gemini。

這個決定純粹是基於「我的情境」，我熟悉 Claude 的回應風格，所以我更容易判斷它的分析結果好不好。這跟技術優劣無關。

後來 EquityHunter 的開發過程中，我確實發現 Claude 在做質化分析（qualitative analysis，就是用文字描述而非純數字的分析方式）上的表現不錯。回測結果也證明了一件有趣的事：純粹用規則去篩選股票其實沒有預測力，但 LLM 做的質化分析反而有參考價值。這是我在決策當下不可能知道的事，但因為我選了一個「我能快速驗證的方案」，我才能在短時間內得到這個結論。

這就是為什麼前面幾章花了那麼多篇幅講知識地圖。你不需要懂技術細節，但你需要有能力「看懂 AI 給你的選項」。看懂的意思是能判斷：這個方案適不適合我現在的情況？

問 AI 問題的技巧

跟 AI 要比較報告的時候，有幾個小訣竅：

第一，明確說出你的情境。不要只問「哪個 API 最好」，要說「我的量大概多少、預算多少、用在什麼場景」。同一個技術在不同情境下，「最好」的定義完全不同。

第二，要求它列出比較維度。價格、效能、維護難度、社群支援……你不需要事先知道該比什麼，讓 AI 建議，你再看有沒有遺漏。

第三，問「你會推薦哪個？為什麼？」AI 的推薦不一定要照單全收，但它的推理過程會幫你看到你沒想到的面向。有時候 AI 會說「我推薦 A，但如果你重視 X，那 B 會更適合」，這種有條件的建議往往最有用。

做技術決策最大的陷阱是追求完美。永遠有更新、更強、更便宜的方案，但你的專案不會等你研究完。選一個夠好的方案，快速驗證，不行再換。實際動手之後你會學到的東西，遠比事前研究多得多。

第 13 章｜CLAUDE.md：教 AI 認識你的專案

想像一下這個場景：你每天早上進辦公室，都會有一個全新的實習生坐在你旁邊。

這個實習生超級聰明、學什麼都快、做事效率也很高。但問題是，他完全不認識你，不知道你在做什麼專案，不知道公司有什麼規矩，上一個實習生踩過什麼坑他也不知道。

每天早上你都得從頭解釋一次。

這就是 AI 的工作方式。每次你開一個新對話，它就是一個全新的實習生。之前的對話內容？不記得了。你上次跟它說過的重要規則？消失了。

CLAUDE.md 就是你寫給這個新實習生的員工手冊。

什麼是 CLAUDE.md

CLAUDE.md 是一個放在專案根目錄的文字檔。每次你用 Claude Code 打開這個專案，Claude 會自動讀取這個檔案，把裡面的內容當成「這個專案的規則」來遵守。

你不需要每次對話都重複說明「這個專案是用什麼技術」「哪些事情不能做」「怎麼跑測試」。寫一次在 CLAUDE.md 裡，AI 每次都會自己讀。

該寫什麼

我不打算給你一個模板讓你照抄，因為每個專案不一樣，硬套模板反而會讓你忽略自己真正需要寫的東西。我講方法論。

你可以從四個方向去想：

專案簡介。這個專案是做什麼的？用了什麼主要技術？部署在哪裡？這些基本資訊讓 AI 一打開專案就有全局觀。

禁止事項。這個等一下我會花比較多篇幅講，因為它比你想像中重要太多了。

特殊規則。你的專案有沒有什麼奇怪的慣例？比如某個資料夾不能動、某個 API 有特殊的呼叫方式、跟別的專案共用資料庫所以表名有前綴……這些「只有你知道」的事情，AI 不會通靈。

怎麼跑測試跟部署。告訴 AI 用什麼指令跑測試、部署到哪個平台、有沒有需要先設定的環境變數。這樣 AI 做完功能後可以自己驗證。

禁止事項比允許事項重要

這是我踩了非常多坑才學到的教訓。

AI 很積極、很主動。你叫它做 A，它常常會「順手」把 B、C、D 也做了。有時候這很方便，但有時候會搞出大問題。

我的 CLAUDE.md 裡面有一條規則，現在看起來理所當然，但當初是被坑過才加上去的：「禁止捏造資料」。

事情是這樣的。有一次我在做一個儀表板，需要顯示一些統計數據。我跟 AI 說「幫我把這個頁面做出來」，它很快就做好了，圖表漂漂亮亮的，數字看起來也很合理。

我差一點就直接上線了。

還好我多看了一眼，發現那些數字不對勁，增長曲線太完美了，每個月的數據呈現一個非常漂亮的線性成長。真實世界的數據不長這樣。

原來 AI 在我沒有提供實際資料的情況下，自己編了一組假數據填進去。它不是故意騙我，它只是在「完成任務」，你要一個有圖表的儀表板，它就做了一個有圖表的儀表板，只是數據是它自己編的。

從那次之後，我在 CLAUDE.md 裡面寫了很明確的規則：「絕對禁止使用假的、mock 的、或自行編造的資料來建構功能。若需要資料但無真實來源，停下來問使用者。」

我甚至加了一條自我檢查機制：「如果你正在寫一個陣列，裡面的數值呈現規律遞增、完美整數、或過於均勻的分佈——你很可能在捏造資料，立即停止。」

這就是為什麼禁止事項這麼重要。你不可能事先想到 AI 會做的每一件好事，但你可以根據經驗，把它做過的壞事一條一條記下來。每次被坑，就多加一條規則。久了，你的 CLAUDE.md 就變成一份經驗累積的防護清單，而 AI 會乖乖遵守。

另一個常見的例子：AI 很喜歡「順手重構」。你叫它改一個小 bug，它改完之後順便把旁邊看不順眼的程式碼也重構了。聽起來很貼心，但這有時候會破壞本來正常運作的功能。所以我的 CLAUDE.md 裡面還有一條：「只做被要求的事，不做沒被要求的事。不要移除看起來不需要的東西。」

你的 CLAUDE.md 不需要一開始就很完整。從三五條規則開始，每次被坑就加一條。幾個專案做下來，你就會有一份非常實用的 AI 員工手冊。

第 14 章｜迭代思維：先求有，再求好

這大概是這本書裡我最想講的一個觀念。

新手用 AI 做產品，最常犯的錯誤就是想一次做到完美。花大量時間描述每一個細節、每一個邊界情況、每一個使用者流程，然後期待 AI 一次就吐出一個完美的成品。

不會的。就算你是世界上最厲害的 PM，你也不可能一次把所有需求都想清楚。因為很多問題是你看到第一版之後才會發現的。

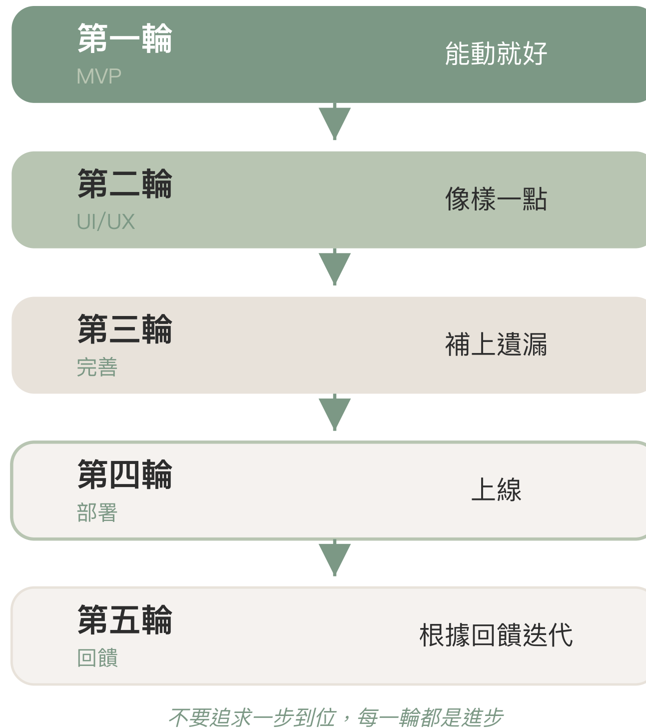
我做 weddingcard-saas（謝卡醬，一個線上做婚禮感謝卡的服務）的時候，第一版醜到我自己都不好意思看。

不誇張。底色是灰的，字體是系統預設，排版歪七扭八，卡片預覽看起來像 2003 年的網頁。我太太看了一眼說：「你確定這個要給人用？」

但它「能動」。使用者可以選一個底色、打字、預覽、下載。功能面來說，核心流程跑得通。

五輪迭代法

我自己的做法是把產品開發分成五輪，每一輪都是一個完整的「目標 → 執行 → 驗收」循環：



圖：五輪迭代流程

第一輪：能動就好

目標很簡單：核心功能可以走完一個完整流程。不要管好不好看、不要管邊界情況、不要管效能。就是能動。

謝卡醬的第一輪：使用者選底色 → 輸入文字 → 看到預覽 → 下載圖片。就這樣。底色只有三個選項，文字排版是 AI 隨便放的，下載出來的圖片解析度也不怎麼樣。

但做完這一輪，我能確認一件最重要的事：這個產品的核心邏輯是通的。使用者確實可以從頭走到尾，產出一張感謝卡。

第二輪：像樣一點

第一輪能動之後，你會很自然地看到最刺眼的問題。不用做什麼優先順序分析，哪裡最醜、最難用，一眼就看得出來。

謝卡醬的第二輪：我加了十個配色方案（本來只有三個很醜的底色）、調整了文字的字體跟排版、讓卡片預覽看起來至少像是這個年代的產品。

這一輪結束後，如果有人問我在做什麼，我至少敢把畫面截圖給他看了。

第三輪：補上遺漏

這一輪你才開始想那些「但是如果⋯⋯」的問題。使用者輸入超長的文字怎麼辦？手機上看起來會不會跑版？下載的圖片能不能更清晰？

謝卡醬的第三輪：我加了文字長度限制、做了手機版的 RWD、提高了輸出圖片的解析度。也是在這一輪加了 Email 寄送功能，做完卡片後可以直接寄給賓客。

第四輪：上線

上線前要處理的事：安全性檢查、金流串接（如果是付費服務）、錯誤處理、基本的效能確認。

謝卡醬是付費服務，所以第四輪花了不少時間在串接 PayUNI 金流。這是必須做好才能上線的東西，不能馬虎。

第五輪：根據回饋迭代

上線之後才是真正的開始。你會收到使用者的回饋，有些你預期得到，有些完全出乎意料。

每一輪迭代之間，你不需要等很久。有了 AI 幫忙，一輪迭代可能只要幾個小時。我做謝卡醬的時候，從第一輪到第四輪上線，大概花了兩個週末。如果我一開始就想做到完美，光是在腦袋裡規劃可能就花掉兩個週末了，然後一行程式碼都沒寫。

你不需要一開始就做對所有事情。你只需要每一輪比上一輪好一點。聽起來很慢？其實比「想清楚再動手」快多了。因為「想清楚」是一個不存在的終點——你永遠覺得還沒想完。

第 15 章｜驗收的能力比指揮的能力更重要

用 AI 做產品，大家都很在意怎麼下指令（prompt）。指令當然重要，但我覺得有一個能力比下指令更重要，而且常常被忽略：驗收。

AI 做完一個東西交給你，你要能判斷它做得好不好。

這聽起來好像很理所當然？但很多人的做法是——不判斷。AI 說做完了，看起來能動，就上線了。

這是很危險的。

一次慘痛的教訓

有一次，AI 幫我做了一個功能，需要呼叫外部 API。做完之後我試了一下，功能正常。但我在 review 程式碼的時候發現，AI 直接把 API Key 寫在前端的 JavaScript 裡面。

API Key 就是你跟外部服務之間的密碼。把它寫在前端，等於任何人打開瀏覽器的開發者工具就能看到你的密碼。有人拿到這個 Key，就可以用你的帳號去呼叫那個 API，帳單算你的，資料安全也完蛋了。

AI 為什麼這樣做？因為它的目標是「讓功能可以運作」，而把 Key 放在前端確實是最快讓功能跑起來的方式。它不是惡意的，它只是在最短路徑完成任務。

從那次之後，我養成了一個習慣：每次 AI 做完一個功能，我都會問它三個「黃金問句」。

三個黃金問句

「有沒有安全風險？」

直接問。AI 通常會很誠實地告訴你它的實作有什麼潛在風險。有時候它會說「目前的實作把 API Key 放在環境變數裡，這是安全的」，有時候它會說「目前的實作沒有做身份驗證，建議加上」。

你要做的是根據它的回答去判斷風險程度，決定哪些現在就要修。

「一百個人同時用會怎樣？」

這個問題會逼 AI 去想效能面的問題。單人用的時候一切正常，但同時有一百個人用呢？資料庫撐得住嗎？API 呼叫會不會超過限制？載入速度會不會變很慢？

AI 的回答通常會分成「現在就有問題的」跟「量大了才會有問題的」。你可以根據自己的產品規模來決定哪些現在就要修、哪些以後再說。

「你有沒有勉強的決定？」

這是我最喜歡的一個問題。AI 在實作的過程中，常常會遇到不確定怎麼做比較好的地方，然後它會自己做一個決定繼續做下去。這個問題就是要把那些「它自己決定但其實不太確定」的地方挖出來。

有時候 AI 會說：「我把資料直接存在本地端而不是資料庫，因為你沒有指定。」或是「我用了比較簡單的驗證方式，但如果是正式環境建議換成更安全的方案。」這些都是重要的資訊。

驗收的重點是使用者角度

我自己有技術背景，看得懂程式碼。但驗收的時候，我刻意不從程式碼出發，而是從使用者的角度走流程。原因很簡單：程式碼正確不等於產品好用。

我的驗收流程通常是這樣的：

先用。打開做好的東西，把每個功能都點一遍。正常流程走一次，異常流程也走一次。故意輸入奇怪的東西、故意按很快、故意用手機開。

再問。用上面三個黃金問句問 AI。

然後請 AI 自我檢查。跟 AI 說：「請你重新檢查一遍剛才的程式碼，有沒有安全問題、效能問題、或是任何你覺得應該改進的地方？」AI 重新檢查的時候，常常會發現第一次做的時候忽略的問題。

最後，如果是要上線的東西，我會特別注意幾個重點：密碼跟金鑰有沒有暴露在程式碼裡、使用者的個人資料有沒有被妥善保護、付款流程有沒有漏洞。

驗收的能力是可以培養的。你做的專案越多，你就越知道該檢查什麼。第一個專案可能只會檢查「功能有沒有動」，到了第十個專案，你會自然地去檢查安全性、效能、錯誤處理。

不管你的技術程度如何，驗收最重要的能力始終是會問問題。把自己放到使用者的位置，問「這裡合理嗎？」這件事，人類比 AI 強多了。

Part 3｜實戰流程 — 從想法到上線

第 16 章｜Phase 1：定義（30 分鐘）

你腦中有一個想法。可能很清楚，可能很模糊，可能只是洗澡時突然冒出來的一句話。

這就夠了。

傳統的產品開發流程會告訴你，你需要做市場調查、寫 PRD（Product Requirements Document，產品需求文件）、畫 wireframe（線框圖，就是產品的草稿圖）、做競品分析。這些在團隊協作的環境裡確實重要，但如果你只是一個人想把腦中的想法變成現實，你需要的只是一段跟 AI 的對話。

我的 ToolBox 工具網站就是這樣開始的。當時的想法真的只有一句話：「我需要一個放常用線上工具的地方。」沒有什麼宏大的商業計畫，就是每次要算所得稅、查勞健保級距、轉換檔案格式，都要 Google 半天找工具，煩死了。

所以我打開 Claude Code，跟它說了大概這樣的話：

> 我想做一個工具網站，把我常用的線上小工具集中在一起。先做一個綜合所得稅計算機就好。用 Next.js，要能直接部署成靜態網站。

就這樣。沒有詳細的功能規格書，沒有資料庫設計圖，沒有 API 規劃。一句話。

你不需要想很完整才能開始

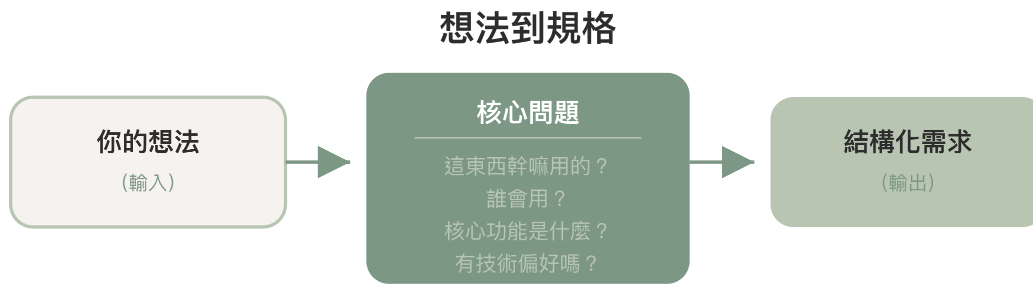
很多人卡在「想法不夠完整」這個心理障礙上。覺得自己還沒想清楚所有的功能、所有的使用場景、所有的邊界情況，所以遲遲不敢開始。

但實話是，你永遠不可能在動手之前就想清楚所有事情。就算你花三個月寫了一份完美的規格書，做出來之後你還是會發現一堆當初沒想到的問題。這不是你的問題，這是所有軟體開發的本質。

所以不如反過來：先做一個最小的東西出來，讓它教你接下來該做什麼。

怎麼把想法變成結構化需求

跟 AI 對話的時候，你不需要用什麼特殊的格式或術語。用你平常說話的方式就好。但有幾個要素如果能提到，會讓 AI 更容易幫你：



模糊的想法經過四個核心問題，變成 AI 能理解的明確規格

圖：從想法到結構化需求的轉換流程

注意第三點：最核心的一個功能。不是三個，不是五個，是一個。

這就是 MVP（Minimum Viable Product，最小可行產品）的概念。你不是要做一個完美的產品，你是要做一個「能用」的東西。能用，意味著它解決了一個具體的問題，哪怕只解決了一個。

ToolBox 的 MVP 就是一個所得稅計算機。一個。不是三十個工具，不是七個分類。就一個計算機放在網頁上，能算出你今年要繳多少稅。做完這個之後，我才開始想：要不要加個勞健保計算？要不要做個匯率換算？

AI 會幫你把模糊的想法變清楚

一個我覺得很有用的特點是，當你跟 AI 說了你的想法之後，它通常會反問你一些你沒想到的問題。比如：

- 「這個網站需要使用者登入嗎？」
- 「資料要存在哪裡？」
- 「你打算怎麼部署？」

這些問題本身就有價值。它們幫你把模糊的想法逼出一個具體的形狀。你不需要每個問題都有答案 — 回答「不知道，你建議呢？」完全可以。AI 會根據你的情境給你建議，你選一個聽起來合理的就好。

三十分鐘之內，你會從「我好像想做一個什麼東西」變成「我要用 Next.js 做一個工具網站，第一個功能是所得稅計算機，部署到 Cloudflare Pages」。

這就是定義階段的全部了。不需要更多。

接下來，開始建造。

第 17 章｜Phase 2：建造（數小時到數天）

建造階段是整個流程中最讓人興奮、也最容易搞砸的一段。

讓人興奮是因為你終於開始看到東西從無到有冒出來。搞砸則通常是因為太興奮了，一次想做太多。

我犯過這個錯。不只一次。

一次一個功能

如果你從這本書只記住一個原則，我希望是這個：一次做一個功能，做完確認沒問題，再做下一個。

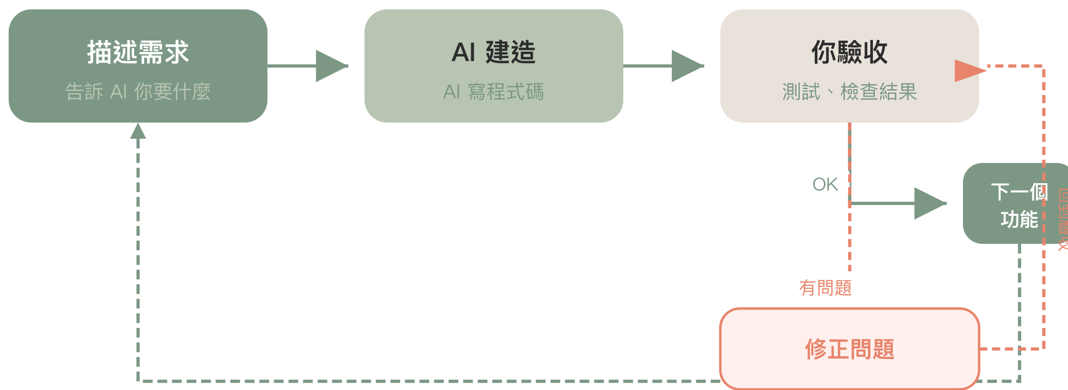
聽起來很直覺對吧？但你會很驚訝自己有多容易違反它。當你興致正高的時候，腦中會不斷冒出新功能的想法：「啊對，這裡應該加一個搜尋功能」「那邊可以做一個排行榜」「如果能支援匯出 PDF 就更好了」。然後你很自然地會想把這些全部一口氣告訴 AI，叫它一次做完。

不要這樣做。

當你一次列出十個功能叫 AI 全部實作，會發生幾件事：首先，AI 會嘗試同時處理太多邏輯，程式碼會變得又長又複雜。其次，如果其中有一個功能做錯了，你很難找到問題在哪，因為所有東西都糾纏在一起。最後，你自己也會搞不清楚目前做到哪了。

正確的節奏應該是：描述 → AI 建造 → 你驗收 → 沒問題就下一個，有問題就修正。

建造循環



描述 → 建造 → 驗收，不斷循環直到滿意

圖：單一功能建造循環

這個循環會一直重複，直到你的產品成形。

怎麼決定功能的順序

ExamBank 考題庫是我做過功能最多的專案之一。105 所學校的考卷、搜尋、下載、上傳、分類、管理後台、Google 登入、社群回報系統 — 功能很多。但它不是一天做出來的。

最初的版本只有一件事：搜尋考卷並下載。

為什麼從搜尋開始？因為搜尋是整個產品的核心價值。如果使用者連考卷都找不到，其他功能都沒意義。Google 登入？之後再說。上傳功能？之後再說。管理後台？之後再說。

決定順序的方法很簡單，問自己一個問題：「如果這個產品只能有一個功能，會是什麼？」那就是你第一個要做的。做完之後，再問：「現在最缺什麼？」答案就是下一個。

ExamBank 的開發順序大概是這樣的：搜尋加下載是核心，然後是分類篩選做體驗優化，接著加 Google 登入為上傳功能準備，再來是社群上傳讓用戶能貢獻內容，之後做管理後台審核上傳的內容，最後是檢舉系統做內容品質控制。

每一步都是在前一步的基礎上自然長出來的。不是我一開始就規劃好這些功能 — 很多是做著做著才發現「啊，這裡需要一個管理介面」。

驗收就是自己用用看

「驗收」聽起來很專業，但其實就是：打開瀏覽器，自己用用看。

點那個按鈕，它有反應嗎？搜尋一個關鍵字，結果正確嗎？在手機上看，排版會不會跑掉？這些你都能自己判斷。重點是站在使用者的角度，看產品該怎麼用。

如果發現問題，跟 AI 描述你看到的狀況就好：「我點了下載按鈕，但什麼都沒發生」「搜尋結果的順序好像不對，最新的應該在最上面」。AI 會去修，修完你再試一次。

這個過程有時候會來回好幾輪。別急，這很正常。專業的軟體工程師也是這樣工作的 — 寫一段、測一段、改一段。差別只是他們自己改，你是請 AI 改。

時間預期

一個簡單的靜態網站（像 ToolBox 的第一版），大概幾個小時就能做出來。一個有前後端的完整應用（像 ExamBank），可能需要好幾天，分成很多個 session 來做。

不要期待一個下午就能做出一個完整的 SaaS 產品。可以做出很多東西沒錯，但「做出來」和「做好」之間有距離。給自己一點時間，享受每個功能從無到有的過程。

第 18 章｜Phase 3：測試（1–2 小時）

我承認，測試是我最初最想跳過的步驟。

心裡的聲音是：「功能都做好了，我自己也試過了沒問題，幹嘛還要寫測試？」這大概是所有開發者（包括很多資深工程師）的共同心聲。

後來踩了幾次坑才學乖。

餐廳的試菜

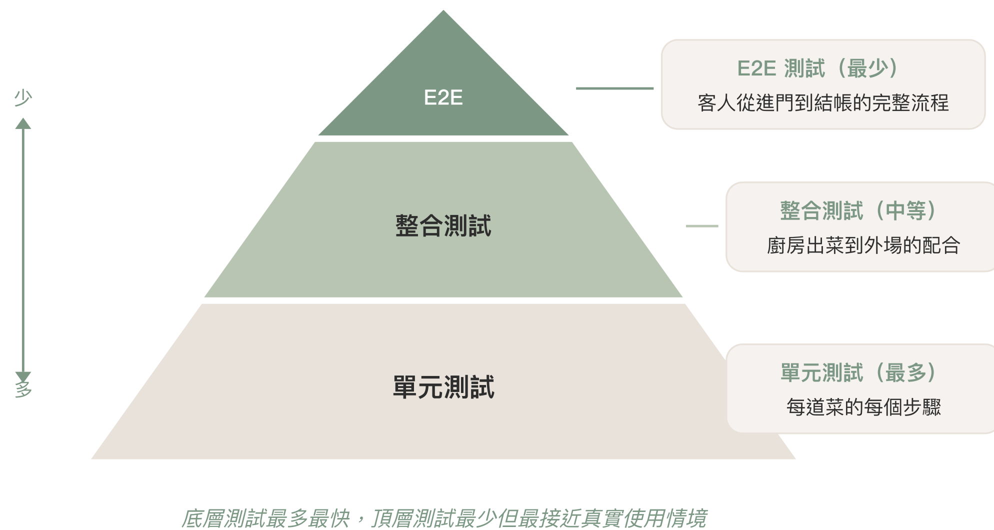
想像你開了一家餐廳。每天開門營業前，你會不會讓廚師先試一下今天的菜，確認味道沒問題？當然會。你不會等客人吃了才發現今天的湯忘了加鹽。

軟體測試就是這個概念。你寫了一堆程式碼，裡面有各種計算、各種邏輯判斷、各種資料處理。你怎麼確保每一個環節都是對的？光靠自己點點按按是不夠的，因為你不可能每次改東西之後都把所有功能重新手動測一遍。

測試程式會自動幫你做這件事。每次你修改了程式碼，跑一下測試，幾秒鐘之內就知道有沒有東西壞掉。

三層測試

軟體測試有三個層次，我用餐廳的比喻來解釋：



圖：三層測試架構

單元測試（Unit Test）是最基本的。它測的是程式裡最小的單位 — 一個函式、一個計算。比如「輸入年收入 80 萬，算出來的稅額是不是正確？」

整合測試（Integration Test）測的是不同元件之間的配合。比如「使用者送出搜尋請求，後端有沒有正確查詢資料庫，然後回傳正確的結果？」

E2E 測試（End-to-End Test）測的是整個使用流程。比如「使用者打開網站、輸入學校名稱、點搜尋、看到結果、點下載 — 整個流程有沒有跑通？」

你不需要自己寫測試

好消息是，這三種測試你都不需要自己寫。

叫 AI 寫就好了。

我的 EquityHunter 股票研究引擎有 295 個測試，涵蓋 80% 以上的程式碼。這些測試我一個都沒有手寫。全部是告訴 AI「幫這個模組寫測試」，它就生出來了。

你可能會想：「AI 寫的測試真的有用嗎？」有。非常有用。因為 AI 寫測試的時候會考慮到很多你想不到的邊界情況 — 輸入是空值怎麼辦？數字是負的怎麼辦？網路斷了怎麼辦？這些你自己手動測的時候通常不會想到，但 AI 會自動幫你涵蓋。

EquityHunter 的測試在每次我修改程式碼之後都會跑一遍。好幾次，我以為「就改了一行而已，不會有問題」，結果測試直接噴紅。去看才發現，改了那一行導致另一個看似無關的功能壞掉了。如果沒有測試，我根本不會知道，直到某天跑出一個莫名其妙的錯誤。

測試不是可選的，是保險

我知道「寫測試」聽起來很無聊，尤其當你只想趕快把功能做完上線的時候。但相信我，花一兩個小時請 AI 幫你寫測試，可以幫你省下未來幾十個小時的除錯（debug，找問題並修復的過程）時間。

你跟 AI 說的話可以很簡單：

> 幫我寫這個專案的測試。單元測試和整合測試都要。覆蓋率至少 80%。

然後去泡杯咖啡。回來的時候，測試就寫好了。

跑一下測試，如果全部通過（通常會顯示綠色），恭喜你，可以安心進入下一階段。如果有紅色的失敗項目，先別慌，跟 AI 說「這幾個測試失敗了，幫我看看是程式有 bug 還是測試寫錯」。通常幾分鐘就能搞定。

第 19 章｜Phase 4：上線（1 小時）

做好了、測完了，然後呢？

讓全世界看到它。

上線（Deployment，把你的產品放到網路上讓別人能用）是很多人最害怕的步驟。因為它涉及到伺服器、域名、DNS（把網域名稱轉換成伺服器位址的系統）、SSL 憑證這些聽起來就很嚇人的東西。

但說真的，2026 年上線一個網站，比你想像的簡單太多了。

我的第一次部署

我第一個上線的 Side Project 是 Wedding — 我跟太太的婚禮邀請函。一個純 HTML/CSS/JavaScript 的靜態網站，部署到 GitHub Pages（GitHub 提供的免費網站托管服務）。

那天晚上我坐在電腦前，跟著 AI 的指示一步步操作。推程式碼到 GitHub、開啟 GitHub Pages、設定自訂域名 wedding.darrenlu.com。

當我在瀏覽器輸入那個網址，看到我們的婚禮邀請函出現在螢幕上的時候——這個東西十分鐘前還只存在我的電腦裡，現在全世界任何人只要輸入這個網址就能看到它。我立刻把連結傳給我太太。

第一次把自己做的東西放上網路，那個感覺確實不錯。

上線前的思考清單

在你興奮到直接按下部署按鈕之前，有幾件事情需要先確認：

```
上線前確認清單
================

[ ] 所有功能正常運作（你自己測過）
[ ] 測試全部通過（綠色）
[ ] 敏感資訊沒有寫在程式碼裡
[ ] 手機和電腦上看起來都正常
[ ] 錯誤訊息對使用者是友善的
    （不是一堆英文程式碼）
```

其中「敏感資訊沒有寫在程式碼裡」這點特別重要，需要多說幾句。

環境變數：你的產品的秘密

你的產品可能需要一些「秘密」才能運作。比如資料庫的密碼、第三方服務的 API Key（就像一把鑰匙，讓你的程式能使用別人的服務）、付款系統的金鑰。

這些秘密不能直接寫在程式碼裡。為什麼？因為你的程式碼通常會放在 GitHub 上，如果秘密寫在裡面，任何人都能看到。就像你把家裡的鑰匙貼在大門上一樣。

Environment Variable（環境變數）就是解決這個問題的方式。簡單說，它是一個獨立的設定檔，放在部署平台上，不會跟著程式碼走。你的程式會說「去環境變數裡找資料庫密碼」，而不是「資料庫密碼是 abc123」。

每個部署平台都有地方讓你設定環境變數。Cloudflare、Vercel、Google Cloud Run 都有。AI 會告訴你怎麼設定。

部署平台怎麼選

不同的產品適合不同的部署平台，但你不需要現在就搞懂所有選項。簡單的對照：

純靜態網頁（沒有後端）適合 GitHub Pages 或 Cloudflare Pages，兩個都免費。有前端加後端的應用適合 Cloudflare Workers、Vercel、或 Google Cloud Run。

如果你不確定，跟 AI 說你的產品做了什麼，它會建議你適合的平台。我自己的專案分散在好幾個平台上 — 靜態網站放 Cloudflare Pages 和 GitHub Pages，需要後端的放 Google Cloud Run 和 Cloudflare Workers。不是一開始就規劃好的，是做到哪就選最適合的。

上線後第一件事

部署成功之後，打開你的手機，在瀏覽器輸入你的網址，從頭到尾把每一個功能都操作一遍。

為什麼要用手機？因為你開發的時候大概率是用電腦，很多問題只有在手機上才會出現。按鈕太小、文字跑出螢幕、某個功能在手機瀏覽器上不能用 — 這些都是常見的問題。

發現問題了就回去修，修完重新部署，再試一次。第一次部署到完全沒問題，通常需要來回兩三次。別氣餒，這很正常。

那個網址一旦活了，你就不再只是「在學做產品的人」。你是一個有作品的人。

第 20 章｜Phase 5：迭代（持續進行）

上線那天你會很開心。

然後大概過兩天，你會開始看到問題。

這不是壞事。這是你的產品開始長大的訊號。

上線不是結束

我做的謝卡醬（weddingcard-saas）是一個婚禮感謝卡的 SaaS 服務。上線的時候，我覺得功能已經很完整了 — 賓客管理、十種色系、信封開啟動畫、QR Code 分享。還做了免費方案和付費方案的區分，甚至串好了 PayUNI 的金流。

然後我把它丟給朋友試用。

朋友 A 說：「我能不能改感謝卡上面的文字？『親愛的某某某，感謝您的到來』這句話我想換成自己寫的。」我沒想到這個。

朋友 B 說：「賓客名單能不能用 Excel 匯入？我有兩百個賓客，不可能一個一個打。」我想到了，但排在後面，現在被提前了。

朋友 C 說：「我想把卡片用 Email 直接寄給賓客，不想一個一個傳連結。」也沒想到。

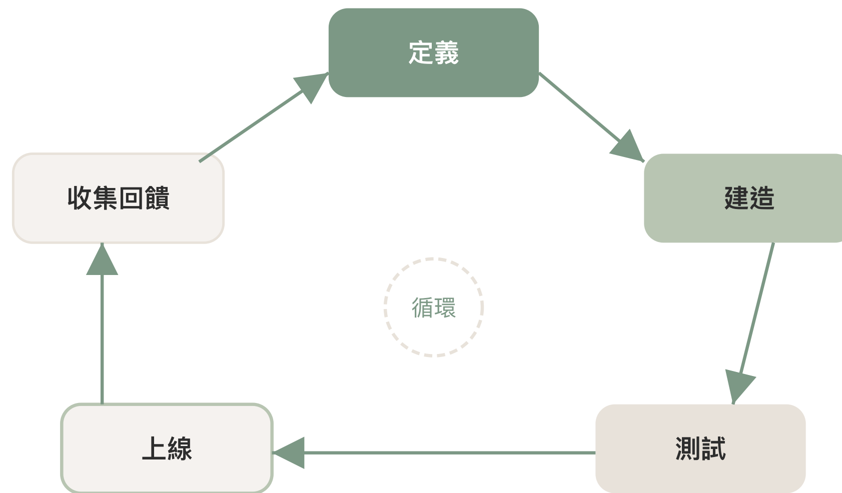
三個朋友，三個我沒預料到的需求。

這就是為什麼上線很重要，只有讓真人用了之後，你才知道接下來該做什麼。坐在電腦前想破頭，也想不出這些東西。

根據真實回饋迭代

迭代（Iteration）的意思是：根據回饋，做一輪改進，然後再收集回饋，再改進。反覆循環。

迭代循環



每一輪循環都讓產品更接近使用者真正需要的樣子

回饋是最重要的燃料 — 沒有回饋就沒有進步

圖：迭代循環流程

關鍵是「根據真實回饋」而不是「根據你的想像」。

在朋友試用之前，我本來計畫的下一個功能是「讓使用者能選不同字體」。聽起來合理對吧？但沒有任何一個朋友提到字體。他們要的是自訂文字、Excel 匯入、Email 寄送。如果我按照自己的想像去做字體功能，就浪費時間在一個沒人在乎的東西上了。

所以謝卡醬後來的開發順序完全照著朋友的回饋走。先做了自訂文字功能，因為改起來最快。然後做 Excel 匯入，因為朋友 B 說「沒有這個我真的沒辦法用」。接著串 Resend 的 SMTP 做 Email 寄送。再後來還加了似顏繪上傳、賓客回饋系統、ZIP 批次匯出。

每一個功能都是因為有人實際說了「我需要這個」才做的。

監控：知道你的產品還活著

產品上線之後，你需要某種方式知道它是不是還正常運作。這叫做監控（Monitoring）。

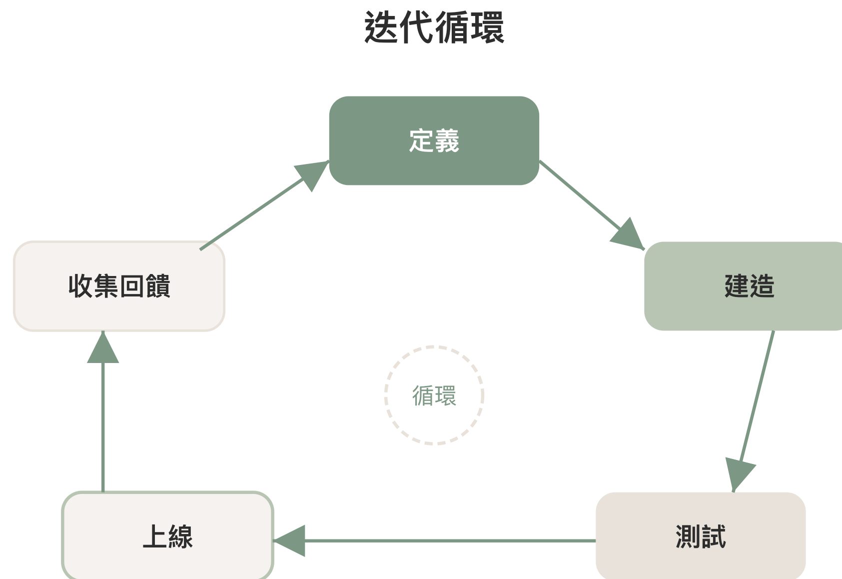
最基本的監控就是：你自己定期去用一下。聽起來很原始，但對 Side Project 來說，這已經夠了。

如果你的產品有後端服務（像是跑在 Google Cloud Run 上的 API），大部分平台都有內建的監控面板，能看到有沒有錯誤發生、回應速度如何、有多少人在用。你不需要自己搭建什麼監控系統，看平台提供的就好。

有一次我的 ExamBank 後端突然收到大量請求，Cloud Run 的面板上錯誤率飆高。我去看了一下 log（系統日誌），發現是某個爬蟲在大量抓考卷。加了 Rate Limiting（限制請求頻率）之後就恢復正常了。如果沒有監控，我可能好幾天後才會發現網站掛了。

永不結束的循環

產品開發沒有「完成」這回事。只有「目前這樣就好」和「下一版要改什麼」。



圖：五階段產品開發循環

這個循環會一直轉下去。每一輪，你的產品會變好一點，你自己也會進步一點。第一輪可能花了你三天做一個基本功能，第五輪的時候你可能半天就能加一個新功能，因為你已經熟悉了整個節奏。

我的 ToolBox 從一個所得稅計算機，長到現在有三十個工具、七大分類。ExamBank 從只有搜尋功能，長到 105 所學校、24 個科目、社群上傳。謝卡醬從一個簡單的卡片生成器，長到有十種色系、Excel 匯入、Email 寄送、似顏繪、賓客回饋。

每一個都是從一個很小的起點開始，經過無數次迭代，慢慢變成現在的樣子。

沒有哪一個是一次做對的。也沒有哪一個是「做完了」的。它們還在長。

Part 4｜心智模型 — 長期受用的思維方式

第 21 章｜懂得怎麼問，比什麼都懂更重要

我在做第一個 Side Project 的時候，大概每十分鐘就會遇到一個需要釐清的整合細節。WebSocket、middleware、CORS、environment variable，這些概念我在學校都學過，但真的要自己從零把它們串在一起的時候，才發現「知道」跟「會用」之間有一段不小的距離。

最直覺的反應是去 Google，然後打開一篇技術文章，讀了三段之後發現，那篇文章預設的情境跟我手上的專案完全不同，照著做反而繞更多路。

這就是第一次獨立處理完整產品時最容易卡住的地方。不是不懂原理，是不知道在自己的情境下該怎麼組合。

但我後來發現一件事：跟 AI 對話，你可以直接從你的具體情境開始問，它會根據你的脈絡給建議，而不是丟一篇通用教學給你。

問對問題比找對答案重要

我踩過最多次的坑，是問錯問題。

舉個例子。我第一次需要處理即時通知的情境，是在做 MemoryTale（一個口述歷史平台）的時候。使用者錄音後，我需要讓前端即時顯示 AI 正在處理的進度。有人跟我說「你需要 WebSocket」。

我第一次問 AI 的方式是：

> 「幫我用 WebSocket 寫一個即時通知功能。」

AI 噴了一堆程式碼給我。我貼上去跑不動，因為我還沒搞清楚在我的架構下 WebSocket 到底該接在哪一層。

第二次我換了個問法：

> 「WebSocket 是什麼？跟一般的 HTTP 有什麼不同？」

這次好多了。AI 告訴我：HTTP 像寄信，你寄一封、對方回一封；WebSocket 像打電話，接通之後雙方可以隨時講話。我重新確認了自己對它的理解。

第三次我再追問：

> 「我的情境是：使用者上傳一段錄音，後端要花 30 秒處理，我想讓前端即時顯示『處理中...80%...完成』。這種情境一定要用 WebSocket 嗎？有沒有更簡單的做法？」

這一次，AI 告訴我其實我的情境用 polling（定時去問伺服器）就夠了，不需要 WebSocket 那麼複雜。

三次提問，從「幫我寫」到「這是什麼」到「我的情境適合什麼」，答案完全不同。

正確問法的公式

我後來歸納出一個很實用的提問順序：

第一步：問「是什麼」。 就算你大概知道，也值得讓 AI 用你當前的情境重新解釋一次。「X 是什麼？在我這種架構下它扮演什麼角色？」

第二步：問「為什麼需要」。「什麼情境下會用到 X？不用 X 會怎樣？」

第三步：問「我的情境適不適合」。把你的具體狀況丟給 AI，讓它判斷。

這三步走完，你對一個概念在你專案裡的定位就夠清楚了。你不需要變成每個領域的專家，你只需要懂到能做出正確決定的程度。

很多人會直接跳到「怎麼實作」，但你連這東西在你的架構裡該放哪裡都不清楚的時候，拿到一堆程式碼也沒用。就像你會開車但第一次自己換變速箱，直接動手不如先搞清楚你這台車的變速箱長什麼樣。

AI 是隨身導師，但你要主動追問

跟 AI 對話跟跟人對話有一個關鍵差異：AI 永遠不會覺得你問太多。

我曾經為了搞懂 CORS（跨來源資源共用）在我的專案裡該怎麼設定，連續問了 AI 大概十二個問題。從「CORS 是什麼」到「為什麼瀏覽器要擋我的 API 請求」到「那我的 FastAPI 後端要加什麼設定」到「如果我前端在 localhost:3000 後端在 localhost:8000 算不算跨來源」。

如果是問同事，問到第五題他大概就會叫我自己去查文件。但 AI 不會。它會一直陪你釐清，直到你真的把問題解決為止。

不過這裡有個陷阱：AI 的回答有時候太有禮貌了。它會給你一個看起來很完整的答案，但其實略過了一些你應該知道的細節。所以你要養成一個習慣，讀完之後問自己：「我真的懂了嗎？還是我只是覺得自己懂了？」

如果有任何模糊的地方，繼續追問。「你剛剛說的 Y 是什麼意思？」「為什麼要先做 A 再做 B，反過來不行嗎？」

即使有技術背景，情境不同就是新的挑戰

我觀察到一個現象：有技術背景的人在用 AI 的時候，反而容易犯一個錯，覺得自己大概懂，就用不太精確的方式描述問題。結果 AI 照著你的假設走，給你一個建立在模糊前提上的答案。

最有效率的做法，反而是把情境交代清楚：「我的專案用 FastAPI 當後端、部署在 Cloud Run 上、前端是 Vite，我現在要處理 X。」

我做了六年的 PM，碰到 Docker、CI/CD、DNS 設定這些東西的時候，雖然概念上都理解，但第一次獨立處理的時候還是會遇到各種環境差異。差別只是我學會了怎麼問，把情境講清楚，讓 AI 給出針對我的狀況的建議，而不是通用教學。

順帶一提，這本書裡會出現一些像 middleware（中介層）這樣的技術詞彙。Middleware 簡單說就是「請求進到你的後端之後、到達你的商業邏輯之前，中間經過的一層處理」，常見的用途是驗證身份、記錄日誌、處理 CORS 等。你不需要自己寫 middleware，但知道它存在，debug 的時候會少走很多冤枉路。

知道自己的情境跟別人不同，才能問出對的問題。問出對的問題，才能得到真正能用的答案。

這大概是我在整個 AI 開發旅程中學到最重要的一件事。

第 22 章｜分辨「AI 該決定的」vs「你該決定的」

用 AI 做產品開發一段時間之後，我逐漸意識到一件事：AI 很能幹，但它不應該幫你做所有決定。

更準確地說，有些決定讓 AI 做比你自己做好得多；有些決定只有你能做，AI 再聰明也不該替你決定。搞混這兩類，要嘛做出技術上糟糕的產品，要嘛做出一個沒有靈魂的產品。

技術決策：放手讓 AI 來

「這個功能應該用 PostgreSQL 還是 MongoDB？」「API 的回傳格式要怎麼設計？」「這段程式碼該怎麼重構比較乾淨？」

這些是技術決策。它們有客觀的好壞標準，有業界公認的最佳實踐，而且 AI 讀過的技術文件比你一輩子能讀的還多。

我剛開始做 Side Project 的時候，會花很多時間自己研究「React 好還是 Vue 好」、「該用 Vercel 還是 Cloudflare Pages 部署」。後來我發現，這些決定交給 AI，描述清楚我的需求（預算、規模、團隊能力），它給的建議通常比我自己研究三天的結論更好。

因為它沒有偏見。我可能因為之前用過某個工具就偏好它，但 AI 會根據你的具體情境推薦最適合的選項。

產品決策：你自己來

「這個 App 的核心功能是什麼？」「免費版要提供到什麼程度？」「要不要收集使用者的個人資料？」「這個功能對使用者真的重要嗎，還是只是我覺得酷？」

這些是產品決策。它們沒有標準答案，取決於你的商業判斷、你的價值觀、你對使用者的理解。

我做 MemoryTale 的時候遇過一個很具體的決定：要不要把使用者的語音錄音存下來？

技術上這很簡單，存到雲端硬碟就好，AI 可以幫我在五分鐘內搞定。但這是一個產品決策，甚至是一個倫理決策。這些錄音是老人家在講自己的人生故事，非常私密。存下來可以讓我們做更好的分析、改善 AI 模型的品質。但如果資料外洩呢？如果使用者根本不想被存檔呢？

我問了 AI：「你覺得我應該存嗎？」

AI 給了我一個很完整的分析，存的優點、不存的優點、如果要存應該怎麼加密、法律上要注意什麼。但它沒辦法幫我做這個決定。因為這牽涉到我對「隱私」的價值判斷、我想經營什麼樣的品牌、我願不願意承擔資料外洩的風險。

最後我決定不存原始語音檔，只保留 AI 轉寫後的文字，而且讓使用者可以隨時刪除。這個決定犧牲了一些技術上的便利性，但我睡得比較安穩。

三類決策對照表

AI 該決定的（技術決策）	一起討論的（灰色地帶）	你該決定的（產品決策）
用什麼框架	功能的技術可行性	要做哪些功能
資料庫怎麼設計	效能 vs 開發速度	功能的優先順序
程式架構怎麼切	第三方服務怎麼選	產品定價
演算法怎麼寫	安全性的程度	隱私政策
API 格式設計	技術債要不要還	品牌定位
部署方式	自己做 vs 用現成的	預算分配
錯誤處理機制	要支援哪些平台	目標使用者是誰

中間那欄「一起討論的」是最有趣的灰色地帶。比如「效能 vs 開發速度」，AI 可以告訴你每個選項的 trade-off，但最終是你根據時程壓力和產品階段來決定。再比如「第三方服務怎麼選」，AI 可以比較功能和價格，但你要考慮的可能還有「這家公司會不會倒」、「它的客服好不好聯絡」這些 AI 不一定能判斷的事情。

一個簡單的判斷法

當你不確定一個決定該自己做還是交給 AI，問自己一個問題：

「這個決定做錯了，我會後悔嗎？」

如果答案是「不會，大不了改就好」，那交給 AI，它可能比你更清楚什麼方案比較好。資料庫選錯了？遷移就好。框架不適合？這個階段換掉成本也不高。

如果答案是「會，而且可能很難補救」——那你自己做。收了不該收的資料、定了錯誤的定價策略、選了不對的目標市場，這些錯誤的代價不是改幾行程式碼就能解決的。

AI 是最強的執行者，但方向盤要握在你手上。

第 23 章｜安全意識與成本意識

雖然我是資工系出身，基本的資安概念都學過，但做了十六個 Side Project 之後，我才真正體會到：知道原理是一回事，在自己的產品上落實又是另一回事。課本上教過 SQL injection、XSS，但真的要自己從零處理 API 金鑰管理、權限隔離、金流整合的時候，才發現處處是細節。

這一章我想把這些實戰教訓整理出來，讓你不用親自踩一遍。

安全：五件刻在腦子裡的事

第一件：API Key 永遠不寫在程式碼裡

這大概是我最接近災難的一次。

有一天晚上我在趕 TutorKit 的進度，需要串接一個 AI 服務的 API。為了快速測試，我直接把 API Key 寫在程式碼裡，反正等一下再移到環境變數就好了嘛。

結果我忘了。

我 commit、push 到 GitHub。十分鐘後我的信箱收到一封 GitHub 的警告信：「你的 repository 偵測到疑似密鑰洩露。」

我當下緊張了一下。雖然那是一個 private repo，而且 GitHub 的 secret scanning 幫我擋住了，但如果那是一個 public repo 呢？有專門的機器人在掃 GitHub 上公開的 API Key，掃到之後幾分鐘內就能被拿去濫用。你的帳單可能隔天就多幾萬塊。

所有的密鑰、密碼、token，永遠放在環境變數（.env 檔案）裡，而且確保 .env 被加進 .gitignore。沒有例外。

第二件：永遠不要信任使用者輸入的東西

使用者輸入的任何東西都可能是攻擊。表單裡填的不一定是名字，可能是一段惡意程式碼。你的使用者不一定是壞人，但你不知道誰會來用你的產品。

你不需要每次都自己手動處理，但你需要告訴 AI：「幫我檢查所有使用者輸入的地方，確保有做驗證和清理（sanitization）。」

第三件：權限要隔離

你的資料庫帳號不應該擁有「可以刪除整個資料庫」的權限。你的前端不應該能直接存取後端的管理功能。每一層只給它需要的最小權限。

我在做 ExamBank 的時候學到這件事。一開始為了方便，前端直接用 Supabase 的 service key（最高權限的金鑰）。後來我意識到，如果有人從瀏覽器的開發者工具把這個 key 挖出來，他就可以對我的資料庫為所欲為。後來趕緊改成只用 anon key 加上 Row Level Security（行級別安全策略），讓每個使用者只能存取自己的資料。

第四件：付款成功不能只看 HTTP 回應

這件事我在做 weddingcard-saas（謝卡醬）整合金流的時候學到的。

你呼叫付款 API，它回你 HTTP 200（成功）。但 HTTP 200 只代表「這個 API 呼叫本身成功了」，不代表「使用者的錢真的付了」。你必須去看回傳資料裡面的交易狀態欄位，那才是真正的付款結果。

200 OK + 交易狀態「失敗」 = 使用者沒付錢。如果你只看 HTTP 狀態碼就把商品發出去，你就是在做慈善。

第五件：上線前問 AI「這個有什麼安全風險？」

每次要部署一個新功能之前，我都會把相關的程式碼丟給 AI，問一句：「這段程式碼有什麼安全風險？」

AI 不一定能找到所有問題，但它能抓到大部分常見的漏洞，像未加密的傳輸、缺少的權限檢查、可能的注入攻擊。養成這個習慣，等於免費請了一個初級資安顧問。

成本：你以為免費，其實有上限

很多雲端服務都有免費方案（free tier），剛開始做 Side Project 的時候，你大概可以一毛錢都不花。但「免費」不是「永遠免費」。

免費方案的甜蜜陷阱

我目前用的免費方案分佈大概是這樣：

服務	免費額度（大概）
Cloudflare Pages	500 次部署/月，頻寬無上限

服務	免費額度（大概）
Supabase	500 MB 資料庫、1 GB 檔案存儲
Vercel	100 GB 頻寬/月
GitHub	私有 repo 免費
Google Cloud Run	200 萬次請求/月（有條件）

這些額度對個人專案或早期 MVP 來說完全夠用。但一旦你的產品開始有真實使用者，數字會開始往上跑。

我的建議是：在什麼都還不確定的階段，用免費方案完全沒問題。但當你的某個指標（使用者數、API 呼叫次數、儲存空間）超過免費額度的 60–70% 時，就該開始規劃付費方案了。不要等到超過了才手忙腳亂。

AI API 的費用最容易失控

這是我最想強調的一點。AI API（不管是 Claude、Gemini、還是 OpenAI）的計費方式是按照你丟給它多少文字（token）來算的。

如果你的產品是「使用者上傳一份文件，AI 幫忙分析」，每一次使用者上傳，你都在燒錢。文件越長，燒越多。

我在做 TutorKit（考卷解析服務）的時候，初期完全沒注意 token 用量。某天去看帳單，發現一個月的 AI API 費用比我想像的高了三倍，因為我在 prompt 裡塞了太多不必要的上下文。

後來我做了幾件事來控制成本：

把不需要最聰明模型的工作，分配給比較便宜的模型。分析考題用好的模型，格式轉換用便宜的模型。

在使用者端做限制：每天免費分析的次數上限、單次上傳的檔案大小上限。

定期去看帳單。這聽起來很基本，但你不看就不知道錢花在哪裡。

不知道多少錢？問 AI

在決定要用某個服務之前，你可以直接問 AI：

> 「我的產品預計有 1000 個使用者，每人每天大約使用 3 次，每次大約處理 500 字的文字。如果我用 Gemini API，一個月大概要花多少錢？」

AI 會幫你估算。數字不一定精確，但至少你有一個量級的概念，是每月 10 塊美金、100 塊、還是 1000 塊。這個資訊會直接影響你的產品決策。

你不需要變成資安專家或財務長

這一章的重點是讓你知道這些風險存在，然後善用 AI 來幫你把關。

安全的部分，養成「每次上線前問 AI 安全風險」的習慣就夠了。成本的部分，養成「每個月看一次帳單、每用新服務前先估價」的習慣就夠了。

這些事情不難，但忘記做的代價可能很大。

附錄 A｜知識地圖速查表

你不需要懂每一層怎麼做，但你需要知道一個軟體產品大概長什麼樣子。

這張圖把一個完整的網路產品拆成六層，從使用者看到的畫面，一路往下到資料儲存。

圖：網路產品六層架構圖

每一層的職責如下：

第 1 層：前端 Frontend — 使用者看到的畫面。HTML / CSS / JavaScript，常見框架有 React、Vue、Next.js。負責「長什麼樣子」。

第 2 層：後端 Backend — 處理商業邏輯。Python / Node.js / Go，常見框架有 FastAPI、Express。負責「怎麼運作」。

第 3 層：資料庫 Database — 儲存所有資料。PostgreSQL / MySQL / MongoDB，服務如 Supabase、Firebase。負責「記住什麼」。

第 4 層：檔案儲存 Storage — 圖片、影片、PDF 等不適合放資料庫的大檔案。Cloudflare R2 / AWS S3 / Google Cloud Storage。負責「放哪裡」。

第 5 層：AI 服務 AI Layer（不是每個產品都需要） — 呼叫外部 AI API，如 Claude / Gemini / OpenAI，做文字生成、圖片辨識等。通常被後端呼叫。負責「聰明的部分」。

第 6 層：部署 Deployment（不是每個產品都需要獨立設定） — 把程式碼放到網路上。Cloudflare / Vercel / GCP，搭配 CI/CD 如 GitHub Actions。負責「怎麼上線」。

每一層你只需要知道一件事：它負責什麼。至於細節，交給 AI。

附錄 B｜跟 AI 對話的黃金問句

以下是我實際在做專案時最常用的問句，按情境分類。你可以直接複製貼上，把 `[...]` 的部分換成你的情境。

情境	問句
開始新專案	「我想做一個 [產品描述]，目標使用者是 [誰]，請建議適合的技術組合，考量我是一個人開發、預算有限、需要快速上線。」
開始新專案	「這個專案用 [A] 還是 [B] 比較適合？請比較兩者的優缺點，以我的情境為基準。」
不懂某個概念	「[術語] 是什麼？請用非工程師能懂的比喻解釋。」
不懂某個概念	「[術語] 跟 [另一個術語] 有什麼差別？什麼時候用哪個？」
技術選擇	「我的情境是 [描述]，用 [A] 還是 [B]？請從成本、效能、維護難度三個角度比較。」
技術選擇	「如果選 [A]，未來要換成 [B] 的成本有多高？」
驗收 / 上線前	「這段程式碼有什麼安全風險？」
驗收 / 上線前	「這個功能上線前，我應該檢查哪些東西？請列出 checklist。」
出問題了	「[錯誤訊息]，這是什麼意思？可能的原因有哪些？從最常見的開始列。」
出問題了	「我改了 [X] 之後 [Y] 就壞了，這兩個有什麼關聯？」
程式碼品質	「請檢查這段程式碼的品質：可讀性、效能、安全性、是否有重複的邏輯。」

情境	問句
程式碼品質	「這個函式太長了，幫我拆成更小的單元，並解釋為什麼這樣拆。」
範圍控制	「我只需要 [核心功能]，不需要 [額外功能]。請不要加我沒要求的東西。」
範圍控制	「這個需求最小可行的做法是什麼？不用完美，能用就好。」
成本估算	「如果我有 [N] 個使用者，每人每天用 [M] 次，用 [服務名稱] 一個月大概多少錢？」
成本估算	「有沒有免費或更便宜的替代方案可以達到類似效果？」

一個額外的提醒

這些問句的重點不是句型，是態度。把你的情境講清楚、把你不懂的地方直說、把你在意的限制（預算、時間、能力）攤開來。AI 拿到的資訊越完整，給你的答案越精準。

附錄 C｜常見的「我不知道自己不知道」

這些是你在做產品之前大概不會想到，但做了之後一定會碰到的東西。

術語	為什麼重要	你要問 AI 的一句話
環境變數 Environment Variables	密碼和金鑰不能寫在程式碼裡，要用環境變數管理，否則洩漏風險極高	「我的專案怎麼設定環境變數？哪些東西應該放進 .env？」
CORS 跨來源資源共用	瀏覽器會擋不同網域之間的請求，不設定的話前端叫不到後端	「我的前端在 A 網址、後端在 B 網址，CORS 要怎麼設定？」
SSL / HTTPS 加密連線	沒有 SSL 的網站，瀏覽器會顯示「不安全」，使用者不敢用	「我的網站怎麼啟用 HTTPS？部署平台有自動提供嗎？」
備份 Backup	資料庫掛了或資料被誤刪，沒有備份就全部消失	「我的 [資料庫服務] 有自動備份嗎？我需要額外設定嗎？」
Rate Limiting 請求頻率限制	不限制的話，有人可以一秒打你的 API 一萬次，帳單爆炸	「我的 API 怎麼加上頻率限制？每個使用者每分鐘最多幾次合理？」
日誌 Logging	出問題的時候，沒有日誌就完全不知道哪裡壞了，只能靠猜	「我的後端要怎麼設定日誌？要記錄哪些資訊才夠除錯？」
版本控制 Git	沒有版本控制，改壞了就回不去。Git 是程式碼的時光機	「教我最基本的 Git 操作：存檔、上傳、回到之前的版本。」

你會在什麼情境碰到它？

環境變數：第一次把專案部署到伺服器的時候。你在本機跑得好好的，上線卻一直報錯，十之八九是環境變數沒設定。每個部署平台（Cloudflare、Vercel、Cloud Run）都有自己的環境變數設定介面，而且本機的 .env 檔不會自動同步上去，你需要手動設定或透過 CLI 工具同步。

CORS：第一次讓前端跟後端分開部署的時候。你在本機開發時前後端可能都跑在 localhost，沒事。但一旦前端部署到 A 網址、後端部署到 B 網址，瀏覽器就會擋住跨來源的 API 請求。你會看到 console 裡一片紅色的 CORS error，然後開始懷疑人生。解法通常是在後端加幾行 CORS 設定，告訴瀏覽器「這個前端網址是可信的」。

SSL / HTTPS：第一次用自己的網域名稱上線的時候。大部分現代部署平台（Cloudflare Pages、Vercel）會自動幫你處理 SSL 憑證，你幾乎不用做什麼。但如果你用 Cloud Run 或自己架伺服器，可能需要手動設定。沒有 HTTPS 的網站，Chrome 會直接在網址列顯示「不安全」，使用者看到就會關掉。

備份：第一次在正式環境誤刪資料的時候。你以為 Supabase 或 Firebase 會自動備份？免費方案通常不會，或者備份的頻率和保留天數很有限。等你發現資料不見了才去查，通常已經來不及了。建議在產品上線前就確認備份策略。

Rate Limiting：第一次發現帳單異常或 API 回應變慢的時候。可能是有人在測試你的 API、可能是爬蟲在掃你的資料、也可能是你自己的前端有 bug 重複發了幾千次請求。加上頻率限制，至少能讓問題不會在你睡覺的時候把帳單燒爆。

日誌：第一次使用者回報「壞了」但你完全無法重現問題的時候。沒有日誌，你只能回覆「我這邊正常」。有了日誌，你可以看到那個使用者在什麼時間點、做了什麼操作、伺服器回了什麼錯誤。差別就像是帶著 X 光片看診，跟只憑「我肚子痛」去猜病因。

版本控制：第一次改壞程式碼想回到上一版的時候。沒有 Git，你唯一的選擇是按 Ctrl+Z 然後祈禱。有了 Git，你可以隨時回到任何一個存檔點。它也是跟 AI 協作的基礎，你可以放心讓 AI 改程式碼，因為改壞了隨時能回退。

這張表你不用一次全學會。當你碰到其中任何一個詞的時候，知道它很重要，然後用附錄 B 的問句去問 AI 就好了。

不知道不可恥。不知道自己不知道，然後上線之後才發現——那才真的會痛。

後記

寫完這本書，回頭看自己這幾年做的十六個 Side Project，有一個感覺很明確：如果沒有 AI，這些東西大概一個都做不出來。

不是因為我不會寫程式。資工系畢業，該學的都學了，工作也做了六年。但 PM 的日常跟「從零到一把產品做出來」是完全不同的事。你在公司裡可以開票、排優先級、跟工程師討論架構。但當你一個人坐在電腦前面，要從 `npm init` 開始，把一個想法變成一個別人打開瀏覽器就能用的東西，那個距離比想像中遠得多。

AI 幫我跨過了那個距離。

但跨過之後我發現，真正讓這些專案能做完、能上線、能有人用的，不是 AI 的能力，是我自己判斷哪些事該做、哪些事不做的能力。AI 可以在十分鐘內幫我搭好一個功能的骨架，但它不會告訴我這個功能到底有沒有人需要。它可以幫我選最適合的技術方案，但它不會告訴我今天晚上應該修 bug 還是去睡覺。

這本書試著把我在這個過程中累積的判斷整理出來。不是技術教學，那個 AI 比我寫得好。是在你已經有 AI 當助手之後，怎麼當一個好的產品負責人。

我知道很多人看完這類書會有一種衝動，想要馬上開始做自己的專案。我的建議是——就去做吧。不用等到準備好，因為你不會準備好。我第一個 Side Project 上線的時候，程式碼亂七八糟，架構也不對，但它能跑，有人在用，那就夠了。後面再慢慢改。

也有些人看完可能會覺得「好像也沒那麼難」。確實沒那麼難。難的不是技術，是持續。是第三個禮拜的時候，進度卡住，使用者數是零，你還願不願意繼續。那個階段 AI 幫不了你，只能自己扛。

我還在扛。每天下班之後打開電腦，繼續做。有些專案活了，有些專案停了。但每一個都讓我學到了什麼，而那些東西最後變成了這本書。

就這樣。去做吧。