AI × DevEx × Platform Engineering

AI時代的平台工程

李智樺



問題：「我們要不要建立 Agent 平台？」

前題：組織有哪些治理能力必須一致提供，才能讓團隊安全地自主開發？

作法：先建立「**最小可行平台**」再**持續迭代**

科技如此進步神速
開發者該何去何從？

AI 讓程式碼的產出速度遠超過人類，
技術工具不斷演進，做法持續被改寫，
在這樣的時代，開發者的價值與方向是什麼？

過去 現在 未來

過去的開發者

以寫程式為核心，
解決技術問題。

- 寫程式
- 除錯
- 用技術解決問題

現在的挑戰

AI 快速產生程式碼，
技術工具日新月異。

- 產出速度更快
- 技術門檻持續降低
- 變化速度超過個人能力

開發者的下一步

不再只寫程式，
而是對產出理解負責。

- 理解需求與業務目的
- 驗證 AI 產出
- 整合資源與約束
- 與團隊協作並持續演化

理解工程

讓理解成為工程化的能力，
是開發者在 AI 時代的方向。

從 DevEx 看理解工程

良好的開發者體驗，應該讓工程師更容易建立、驗證並共享對系統的理解。

回饋迴路 Feedback Loops

快速驗證理解

認知負荷 Cognitive Load

降低理解成本

心流狀態 Flow State

持續深化理解

理解工程 Understanding Engineering

將知識、資源與約束整合為
可運行的現實系統

DevEx 不應只關心工程師能否順利完成工作，**也應關心工程師是否能在工作過程中建立足夠的理解。**

對工程師而言，平台工程最重要的是什麼？

- 平台工程的價值，是讓工程師不必掌握所有技術細節，仍能理解系統如何運作、判斷變更的影響，並在必要時介入與負責。
- 傳統平台工程透過「內部開發者平台」 IDP, Internal Developer Platform 來降低工程師在基礎設施、部署、環境設定與工具整合上的負擔。
  - 降低認知負荷，不等於提高系統理解。
- AI 時代，平台工程還有另一項重要任務：

適當隱藏複雜性，但不要讓理解系統所需的資訊也跟著消失。

平台的關鍵價值

從傳統軟體開發到 AI Agent 時代，工程師面臨的變化



	傳統軟體開發	AI Agent 時代
程式碼	工程師撰寫	AI Agent 產生
理解的主要來源	親自設計與實作	需求、架構、紀錄與驗證 ②
開發過程	相對容易追蹤	多個 Agent 可能同時進行 ①
主要挑戰	如何正確實作	如何掌握 AI 做了什麼 ③
平台的關鍵價值	降低開發摩擦	降低開發摩擦 並**維持系統可理解性**

接下來的討論

1 多工

多個 Agent 併行，
必然產生新的多工模式

2 Code Review

如何在 AI 產出的變更中
快速判斷與把關

3 不看 Code 能理解

建立足夠的系統理解
才能勝任維護並負責

平台工程師

從 UX 到 DevEx 的角度來看

不是讓工程師理解平台，而是讓平台充分理解工程師

傳統平台思維
平台能提供什麼？



DevEx 思維
以工程師為中心

觀點	傳統平台思維 以平台為中心		DevEx 思維 以工程師為中心
設計起點	平台能提供什麼	→	工程師需要完成什麼？
關注焦點	工具、功能、自動化	→	工作情境、阻礙、認知負荷
設計方式	建立標準流程	→	理解工作流程並減少摩擦
成功標準	平台功能完整？	→	工程師能順利完成工作
核心問題	平台是否好用？	→	工程師的困擾是否真正被解決

好的平台，不是要求工程師適應平台，

而是平台能理解工程師的工作情境、困擾與意圖，進而提供適當的協助。

AI 時代的開發者體驗

DEVELOPER EXPERIENCE IN THE AI ERA

01 **必然性的多工**
AI 正在改變開發者的角色與工作型態

02 **不看 code 如何理解？**
從程式碼到意圖、決策與系統理解的新方法

03 **讓「理解」成為良好的開發者體驗**
以工具、流程與團隊實踐，建立可解釋、可信任的 AI 協作

04 **AI-native Software Development Lifecycle**
從需求、設計到維護，重新定義 AI 時代的開發生命週期

05 **AI Governance**
在創新與風險之間，建立負責任的 AI 開發治理



1

必然性多工？



談 必然性多工

不會乖乖等待，
工程師往往同時交辦多個任務，因此產生必然性的多工。



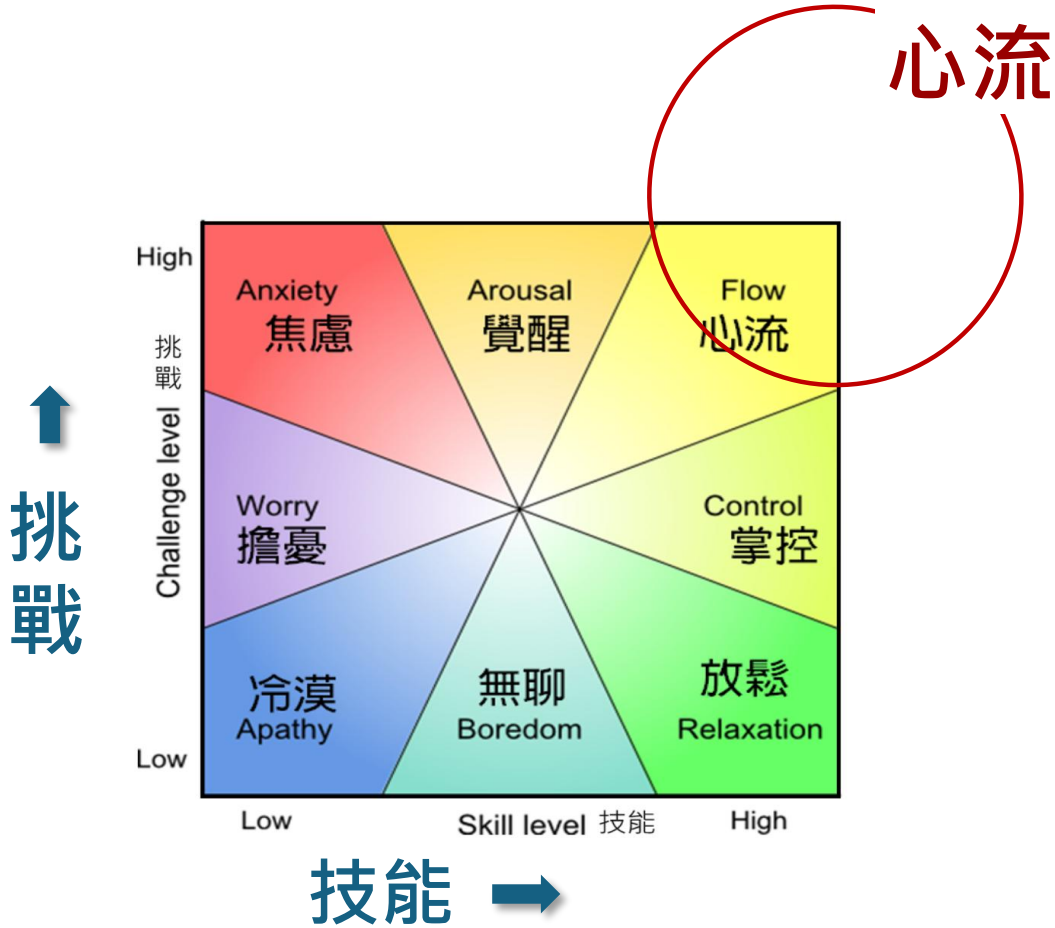


追求心流

- 米哈里·契克森米哈伊
Mihaly Csikszentmihalyi

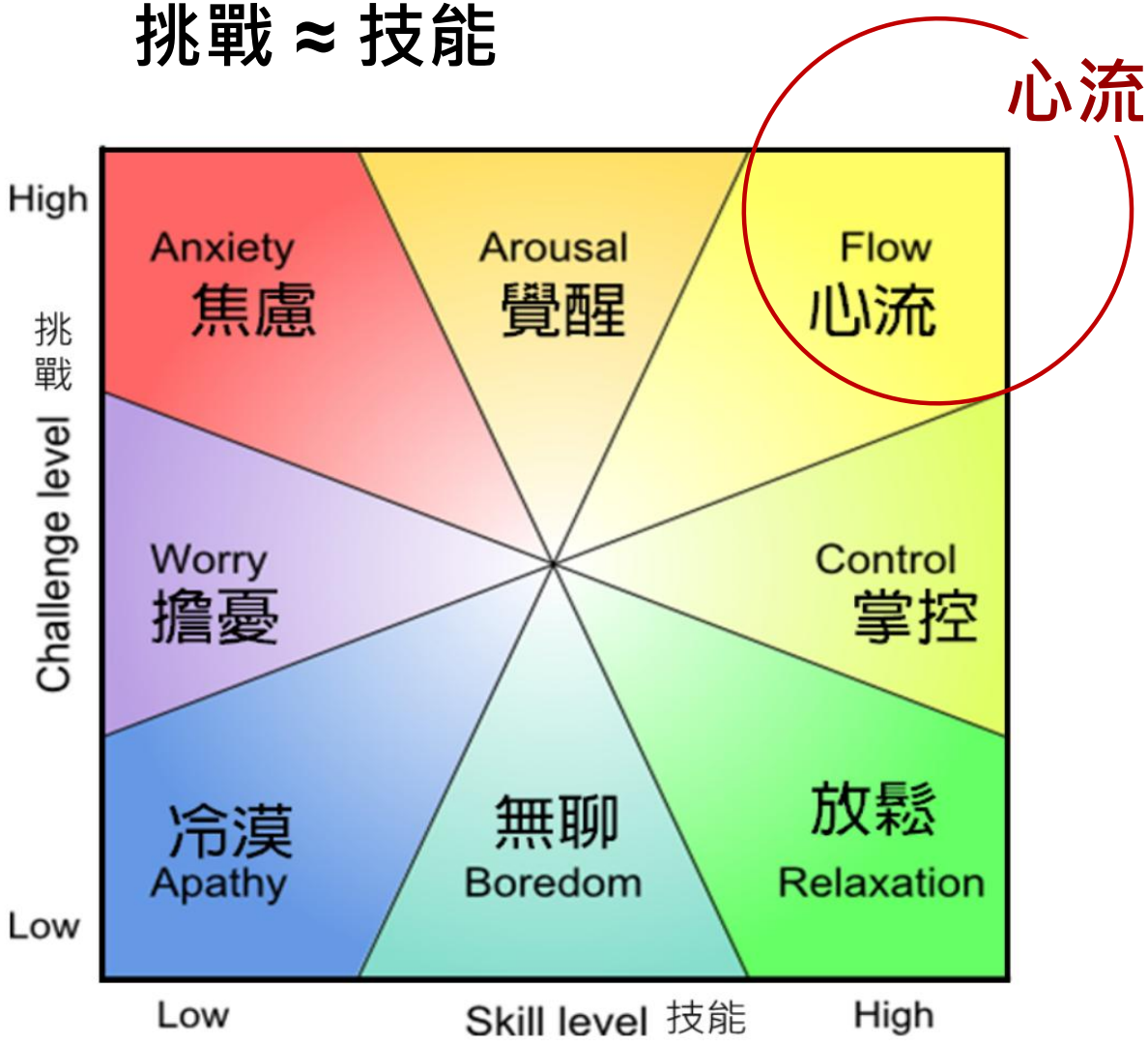
開發者體驗的幸福模型 FLOW

挑戰 ≈ 技能

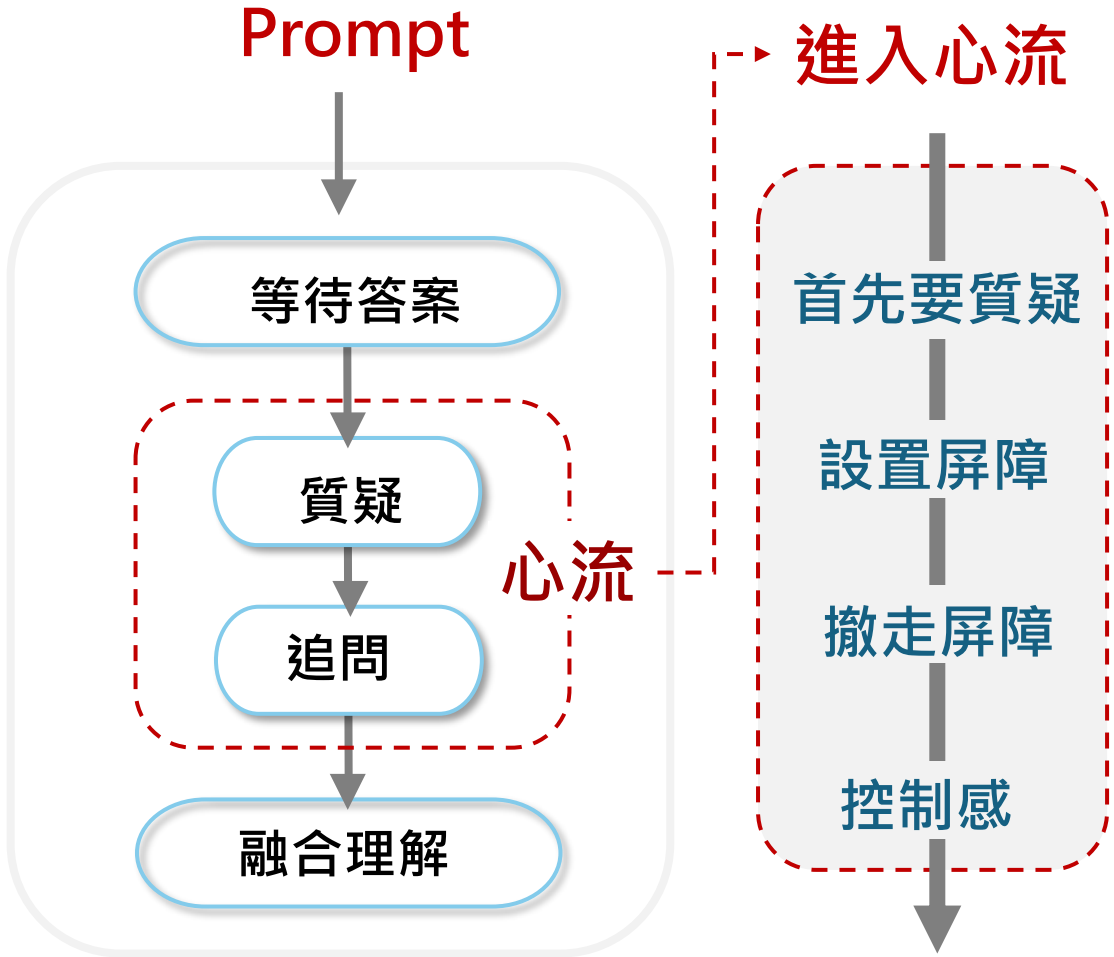


另類心流（Flow）

挑戰 ≈ 技能



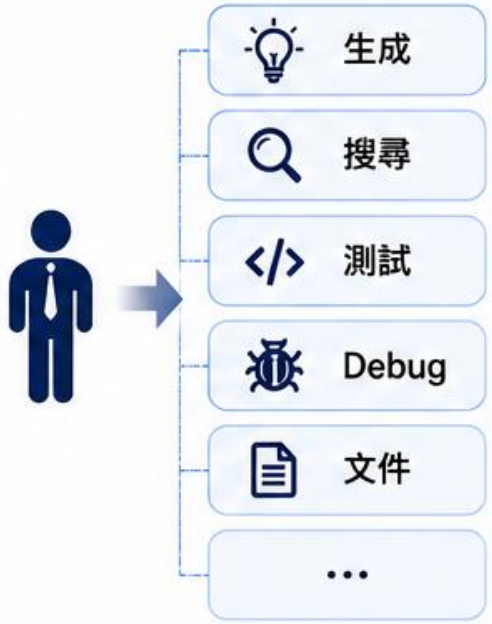
下 Prompt時如何進入心流



AI 時代的工作演化路徑

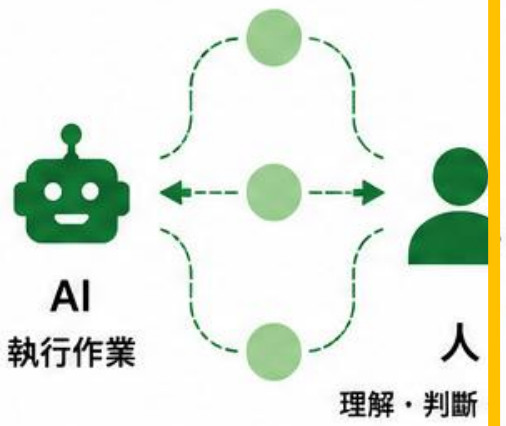
① 工作拆解 → ② 工作重組 → ③ 多脈絡單一專注 → ④ Dynamic Focus

AI 取代部分工作



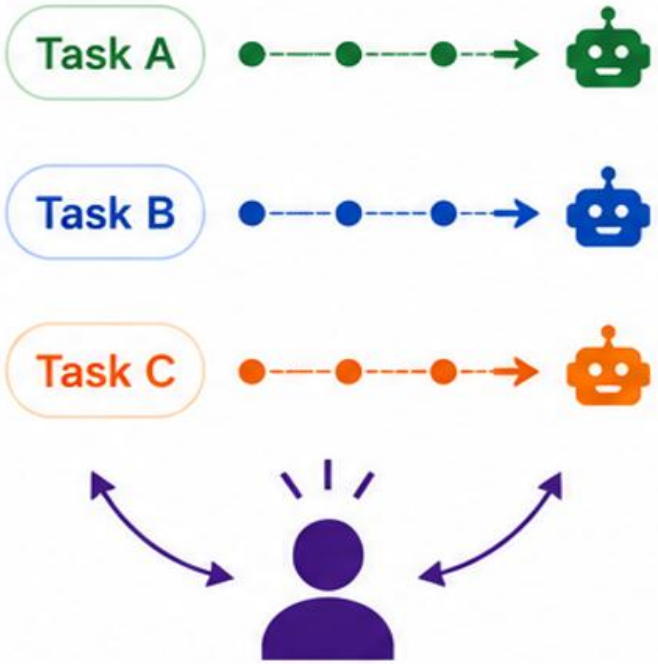
工作被拆解
AI 承接合適部分

重新安排人與 AI



AI 平行處理
人專注關鍵決策

管理多個工作脈絡



人一次專注一個
在多個脈絡間切換

動態專注循環

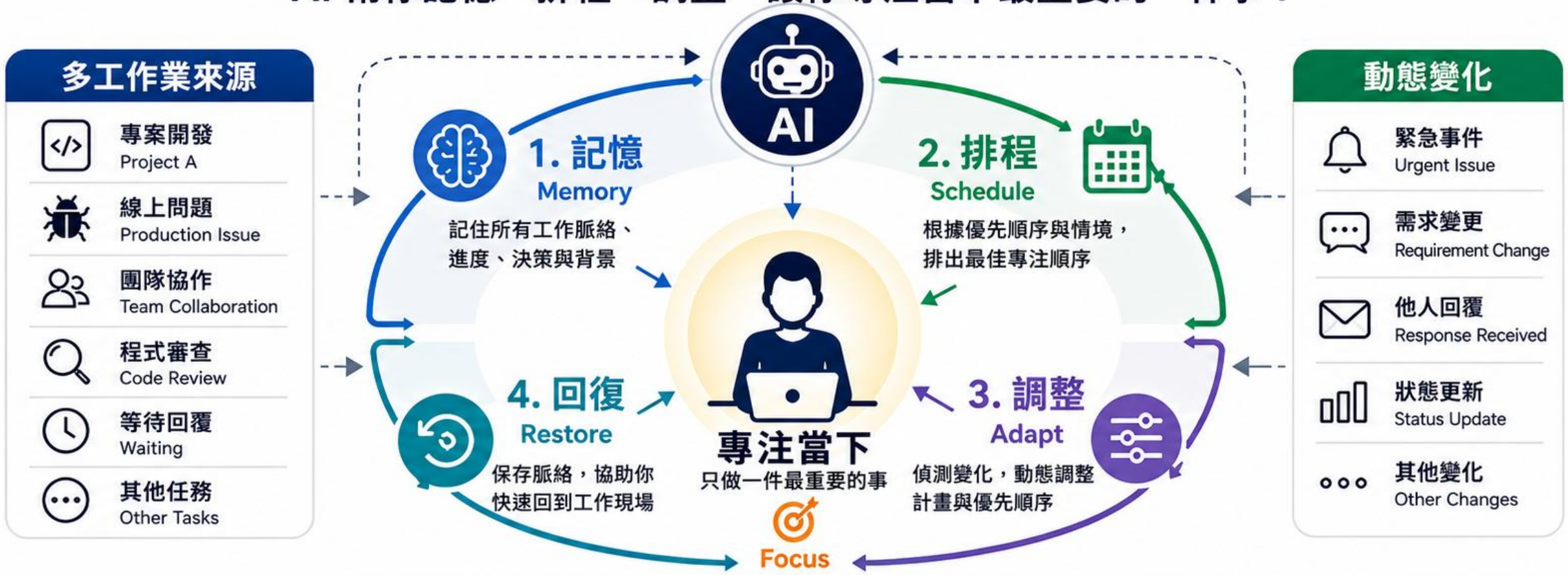


AI 保存脈絡
快速恢復專注

工作重組

動態專注框架 Dynamic Focus Framework

AI 幫你記憶、排程、調整，讓你專注當下最重要的一件事！



AI 產生的專注路徑（你的課表）

開始 Start → 專案開發 Project A → 線上問題 Issue → 程式審查 Review → 專案開發 Project A → 團隊協作 Collaboration → ... → 結束 End

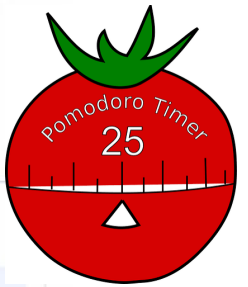
核心價值

讓 AI 管理複雜度，讓你保持專注與深度思考，
在快速變動中持續產出高品質的成果。

人：專注與決策　　AI：記憶與調度

AI 多工功課表（3 Tasks 範例）

工作 → 封存 → 切換 → 載入 → 恢復 → 工作



時間	09:00 - 11:00	11:15 - 13:00	14:00 - 16:00	16:15 - 17:45	18:30 - 20:00
專注任務	Task A 核心模組開發	Task B 跨團隊評審會議	Task C 緊急修復 (Bug)	Task A 延續開發	Task B 文件整理與追蹤
任務類型	開發（深度專注）	協作（溝通協調）	緊急處理（快速應對）	開發（深度專注）	協作（整理彙整）
主要目標	完成模組 X 功能實作	完成架構評審與 決策共識	定位並修復 生產環境 Bug	延續模組 X 測試與優化	整理會議紀錄與 追蹤行動項目
課表安排	120 分鐘 深度專注時段	105 分鐘 會議與討論	120 分鐘 專注修復時段	90 分鐘 深度專注時段	90 分鐘 整理與追蹤
AI 協作重點	• 需求澄清 • 程式碼建議 • Debug 協助	• 會議重點整理 • 決策紀錄 • 行動項目追蹤	• 問題分析 • 修復方案建議 • 影響評估	• 測試案例建議 • 效能優化建議 • 程式碼 Review	• 文件彙整 • 進度追蹤 • 後續提醒設定

任務清單

- **A** 核心模組開發 — 高優先・高專注
- **B** 跨團隊協作 — 中優先・需溝通
- **C** 緊急修復 (Bug) — 高優先・即時處理

排課原則

- A 為核心，安排最長的深度專注時段
- B 安排在上午與傍晚，利於團隊配合
- C 優先處理，穿插在專注時段之間
- 每次切換都執行「切換儀式」
- AI 協助封存脈絡，降低切換成本

切換儀式

（60~90 秒）

1. 工作 — 專注當前任務
2. 封存 — AI 封存脈絡卡片
3. 切換 — 決定下一個焦點
4. 載入 — 載入新任務脈絡
5. 恢復 — 快速進入狀態
6. 工作 — 開始下一個任務

持續循環

AI 脈絡卡片範本

- 任務目標
- 已完成
- 下一步行動
- 關鍵脈絡 (變數、決策、TODO)
- 當前假設 / 思考方向
- 潛在風險與注意事項

多工轉換的 四個階段

經驗

從專注一件事 → 壓縮脈絡 → 快速回來 → 動態多工，讓效能最大化

Level 1 先學會專注 Main Focus

先不要談 AI 多工，先回答：我現在在做什麼？

▶ 每天選一件：Main Focus

- 建立清晰的當下目標
- 減少分心，累積深度產出

Level 2 學會離開 Context Compression

學會好好的「結束一段工作」，回答：我做到哪裡？我現在怎麼想？
下一步是什麼？（用 30~60 秒作回答）

- 把工作脈絡壓縮成精華
- 降低重啟成本，釋放心理負擔

Level 3 學會回來 Context Recovery（縮短 Time to Focus）

回來最重要：回到 Task A 時，不是從頭閱讀。
而是讓 AI 告訴你：① 你上次停在哪裡 ② 原假設是什麼
③ AI 在你離開後完成了什麼 ④ 下一步建議 / 需要你決定什麼
目標不是記憶。而是：縮短 Time to Focus。

- 快速獲得脈絡全景
- 縮短重啟時間，回到高效狀態

Level 4 開始動態多工 開始：A → B → C → A

到了這一層，才開始動態多工：A → B → C → A

▶ 記住：每一次切換都有理由，不是隨機跳來跳去。

- 善用等待時間，AI 並行執行
- 人類動態選擇，保持決策品質

核心價值

保持思考連續性
不論離開多久，都能快速回到狀態

提高交付效能
減少重複理解，專注創造價值

降低壓力與焦慮
AI 幫你記住，讓你安心專注

即使沒有多工需求也有用
每天壓縮與回顧，持續提升效能

AI 脈絡重啟卡

NOW	我在哪？	做到哪裡、卡在哪裡
THINK	想甚麼？	當前假設、思考方向
NEXT	下一步？	回來後要做什麼
WATCH	注意？	風險、別忘記的事

工程師只需要說

- **NOW** ：DB 連不上已排除
- **THINK** ：可能是 Connection Pool 出問題
- **NEXT** ：比較 Production 設定
- **WATCH** ：尖峰流量時才發生

2

Code Review

不看 code 如何理解?

理解，就是讓



我以為系統怎麼運作

逐漸接近



系統實際怎麼運作

Read the Code → Read the System

心智模型 Mental Model

Code Review | 五指理解法

理解到位，才有依據批准 AI 的變更

差距

知識 . 環境
習慣 . 動機
技能 · 溝通



1	拇指 | 變更	我知道 AI 改了什麼？
2	食指 | 結構	我能解釋系統如何運作？
3	中指 | 風險	我知道哪裡可能失敗？
4	無名指 | 證據	失敗時，我知道去哪裡找證據？
5	小指 | 預測	我能預測修改後的行為？

五指合攏，形成足以負責的判斷

Code Review

工程師如何 Read the System？

讓心智模型，逐漸接近系統的實際運作

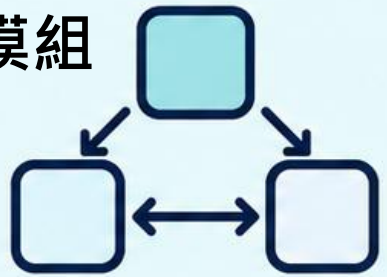
需求



看變更

AI 改了什麼？
影響哪些行為？

模組



看結構

模組如何相連？
資料如何流動？

漣漪



看風險

哪裡可能失敗？
影響會擴散到哪？



看證據

測試與監控，
是否支持我的判斷？

足夠理解 → 有依據批准 → 能為決定負責

Human in the loop

不看 Code，也能理解

先建立心智模型，再用證據驗證

1 **WHY**
為什麼改？



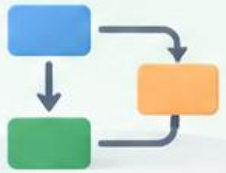
需求模型

2 **WHAT**
改了什麼？



結構模型

3 **HOW**
怎麼運作？



行為模型

4 **WHAT IF**
如果出錯？



失敗模型

5 **EVIDENCE**
證據在哪？



真實模型



我的心智模型

逐步逼近系統的真實行為



真實的系統

Code Review｜五指理解法

理解到位，才有依據批准 AI 的變更

工程師只有在達到了一定程度的理解
才可能勝任維護的工作，才可能負責



1. **拇指｜變更**
   我知道 AI 改了什麼？
2. **食指｜結構**
   我能解釋系統如何運作？
3. **中指｜風險**
   我知道哪裡可能失敗？
4. **無名指｜證據**
   失敗時，我知道去哪裡找證據？
5. **小指｜預測**
   我能預測修改後的行為？



1 接觸表面（拇指）

只知道 AI 修改了什麼

- 看過變更摘要
- 知道影響哪些檔案
- 對背景與目的不清楚



2 理解運作（食指）

能解釋系統如何運作

- 理解主要流程與結構
- 知道關鍵模組與相互關係
- 能說明為何這樣設計



3 理解影響（中指）

知道變更的風險與影響

- 能識別潛在問題點
- 理解對周邊系統的影響
- 考慮效能、安全性等風險



4 找到證據（無名指）

能定位問題並驗證

- 知道如何收集與閱讀證據
- 能使用日誌、監控、測試
- 能重現問題並分析原因



5 預測行為（小指）

能預測修改後的行為

- 能推演 AI 產出的結果
- 評估不同情境的影響
- 提出改進建議與替代方案



不足以維護

只能做初步檢視
需要他人指導

能勝任維護

可以獨立進行維護
並評估變更影響

可以負責

能主動優化、承擔責任
面對問題並提出解決方案

達到此程度 → 能勝任維護

- 理解運作方式、影響範圍與風險
- 能處理常見問題、進行合理變更
- 有能力維持系統穩定運作

達到此程度 → 才能負責

- 能預測後果並採取適當行動
- 能在問題發生時主導解決
- 願意並能對系統長期發展負責

將需求拆解成多種模式

從不同視角理解同一個需求，形成完整的系統圖像

同一個需求，
可以用不同的模式來理解，
每個模式都回答不同的問題。

使用者需求 (Problem)

使用者想要
更快速查詢訂單狀態

→ 拆解

→ 用不同模式看問題

建立完整理解

→ 指導設計與實作

可落地的解決方案 (Solution)

被充分理解、可驗證、可維護

同一個需求，用不同的模式拆解

1 行為模式 Behavior Pattern

系統如何回應
使用者或事件？



- 使用者旅程
- 使用案例
- 狀態機
- 業務規則

描述系統應有的行為

2 互動模式 Interaction Pattern

元件如何合作
完成工作？



- 互動流程
- API 介面
- 事件與訊息
- 時序圖

描述元件之間的協作

3 責任模式 Responsibility Pattern

誰負責決策？
誰負責執行？



- 領域模型
- 模組責任
- 角色與權限
- 決策規則

描述各元件的職責

4 依賴模式 Dependency Pattern

系統如何耦合？
改變如何傳播？

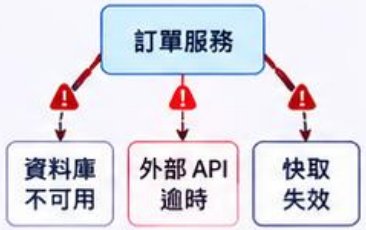


- 相依關係圖
- 資料流向
- 耦合點分析
- 變更影響評估

描述系統的相依關係

5 失效模式 Failure Pattern

哪些地方可能失敗？
如何復原？



- 失效情境
- 邊界條件
- 降級策略
- 監控與告警

描述可能的風險與應對

這些模式彼此關聯、互相驗證、共同形成對需求的完整理解

由個人理解到共同知識

將理解缺口回饋到模型，形成持續演進的 Living Documenitation

1 **WHY**
為何要改？
需求模型
Impact Mapping

2 **HOW**
如何運作？
系統架構圖
C4 / 資料流程

3 **WHAT**
改了什麼？
行為模型
BDD / 狀態轉換

4 **WHAT IF**
可能出什麼問題？
風險模型
故障情境 / 邊界分析

5 **EVIDENCE**
如何確認是真的？
測試模型
可觀測性 / Log / Trace

AI
產出變更

- Code
- 說明文件
- 測試案例
- 架構圖
- ...

Code Review：五層理解檢驗（個人）

WHY 動機	HOW 結構	WHAT 變更	WHAT IF 風險	EVIDENCE 證據
我知道為何要改嗎？	我知道如何運作嗎？	我知道改了什麼嗎？	我知道哪裡會出問題嗎？	我有足夠證據相信嗎？

發現理解缺口或新的問題

理解缺口
回饋到模型

更新與補強（個人 → 團隊）

- 向 AI 提問補充說明
- 補充文件與範例
- 新增測試與驗證
- 更新架構與決策紀錄
- 整理為可重複使用的知識

理解足夠

團隊共同知識

被保存、共享、持續演進

Living Documentation

- 需求 (Why)
- 架構 (How)
- 行為規格 (What)
- 風險與決策 (What If)
- 測試與觀測 (Evidence)
- 經驗與範例 (Know-how)

持續演進
更少重複問題
更快的理解
更高的系統可信度

持續回饋，深化理解

敏捷時代的五個拆解層次

階段	核心問題	主導角色	主要產出
需求拆解	為什麼需要？ 有哪些假設？	PO 主導，團隊共同探索	業務意圖、需求假設、業務情境、業務規則、驗收條件
設計拆解	系統如何承擔責任？	工程師主導，PO 參與確認	領域模型、設計決策、系統責任
工作拆解	如何可靠地完成？	工程師主導	Tasks、依賴關係、完成條件
增量整合	系統是否正確運作？	工程師主導，團隊共同驗證	Working Increment、測試結果
回饋調適	是否真正創造價值？	PO 與團隊共同主導	學習成果、Backlog 調整

Code Review｜五指理解法

理解到位，才有依據批准 AI 的變更



1	拇指｜變更	我知道 AI 改了什麼？
2	食指｜結構	我能解釋系統如何運作？
3	中指｜風險	我知道哪裡可能失敗？
4	無名指｜證據	失敗時，我知道去哪裡找證據？
5	小指｜預測	我能預測修改後的行為？

五指合攏，形成足以負責的判斷

差距

工程師不看 Code，要看什麼？

不看程式碼，而是從**多個視角理解系統**，建立可驗證、可承擔責任的判斷。

1 **意圖**
WHY
為什麼要做？
解決什麼問題？
看需求與目的

2 **行為**
WHAT
系統會做什麼？
如何回應？
看行為與規格

3 **結構**
HOW
如何運作？
元件如何協作？
看架構與互動

4 **風險**
WHAT IF
可能出什麼問題？
如何應對？
看風險與邊界

5 **證據**
EVIDENCE
怎麼知道是真的？
可以信任嗎？
看測試與觀測

五層理解模型
建立完整的系統理解

Code Review
個人理解檢驗

持續的理解迴路
AI SDLC × Living Documentation

Human-in-the-loop
工程師的責任與判斷

從理解到信任，持續演進

3

讓「理解」成為良好的開發者體驗

DevEx 觀點

IN THE AI ERA

過去，我們常視為：
如何讓工程師
更有效率地完成開發工作？

AI 時代，增加另一個問題：
如何讓工程師在完成工作的過程中，
持續累積對系統的理解，
並讓這些理解成為團隊的共同能力？

不要只優化產出的速度，
更要優化**理解形成的過程**。



AI 時代

詮釋理解工程

從 DevEx 出發，讓理解更容易形成、能夠驗證、持續深化，成為團隊的共同能力。

DevEx 構面	AI 時代理解工程的關注	實踐策略
回饋迴路 Feedback Loops	多快知道自己的理解是否正確？	**假設驅動驗證** 提出假設，驗證理解，快速修正
認知負荷 Cognitive Load	理解 AI 產出需要多少認知成本？	**模組化與脈絡工程** 降低複雜度，提供清楚脈絡
心流狀態 Flow State	工程師能否持續深化對系統的理解？	**學會心流與動態專注** 保持專注，持續深化理解



問題 與 回答

歡迎提問，我們依流程進行回應



問題 10

分析 3

回答 1

確認 3

完成

High

Low

請寫下你想問的問題
並貼在問題區

問題描述

加以重點
描述

解答

在一分鐘內
做解答

點頭

提問者
確認解答

最後的
回顧檢視

專案開始之初，首重看見全貌

一旦；當你把眼光投注在哪一個要項的時候，
實際上你就只看到那一部分，

你的思緒將被那一部分的內容所牽動，很難再看見其他的事...
所以我們要退後一步，

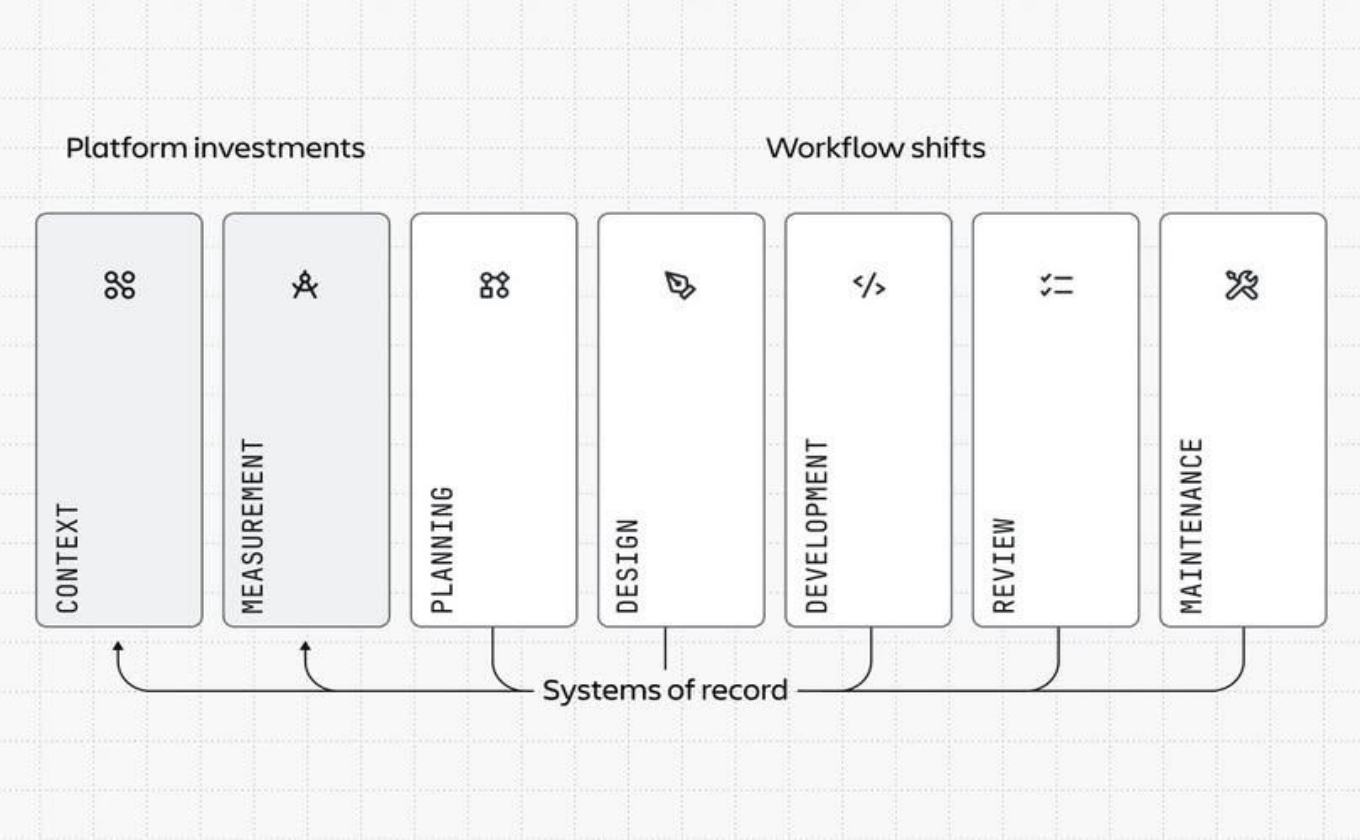
不! 有時要退後很多步，才能比較清晰地看見全貌。

- 請思考你的「**退後一步**」是什麼?

4

Atlassian

《The AI SDLC Transformation Playbook》



Atlassian 艾特萊森軟體公司 資深副總裁 Matt Schvimmer 2026 發布

SDLC

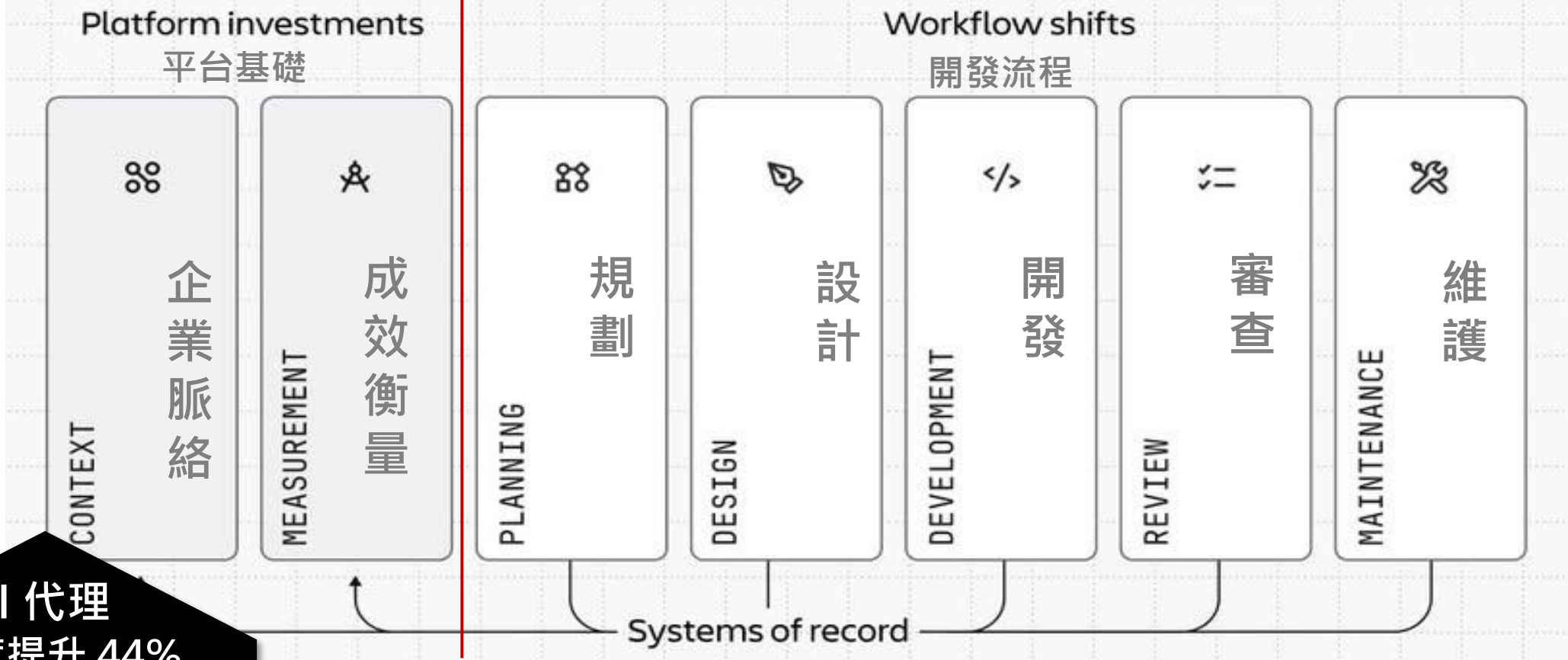
Atlassian

《The AI SDLC Transformation Playbook》

Platform Foundation ⇄ Adaptive Workflow ⇄ Continuous Learning

敏捷

開發平台的基礎能力 ← → 持續互動、反覆調整的五種開發活動



AI 代理
準確度提升 44%,
Token 消耗少了 48%

工作協調者 ➡ 工作設計者

軟體開發的核心議題

第一性原理

工程師的核心工作，是持續深化對專案的理解
並將其轉化為值得信任、能創造價值的軟體

面向	核心問題	可觀察的證據
價值	為誰解決什麼問題？ 為何值得做？	目標、優先順序、成功條件
實現	能否把理解轉為 可運行、可修改的系統？	從想法到可驗證成果的阻力與時間
可信	憑什麼相信系統符合預期， 且風險可接受？	驗收範例、測試、實際運行證據
學習	結果是否支持原先的理解？ 下一步該怎麼改？	從回饋到調整、再驗證的速度

《The AI SDLC Transformation Playbook》

Atlassian

敏捷

Platform Foundation
打造支援 AI 的開發基礎

Adaptive Workflow
人機協作的敏捷開發流程

Continuous Learning
持續學習與系統演化

01	02	03	04	05	06	07
CONTEXT 脈絡	MEASUREMENT 衡量	PLANNING 計畫	DESIGN 設計	DEVELOPMENT 開發	REVIEW 檢核	MAINTENANCE 維護
建立 AI 能理解的系統、業務及組織脈絡	建立衡量開發效能、品質與成果的能力	需求探索、規劃與工作拆解	架構分析、設計與方案比較	程式開發與測試	程式審查與驗證	維護、除錯及系統演化

Systems of record

貫穿所有階段的權威紀錄系統，保存知識、決策、程式碼、測試、部署與營運資料

平台工程，讓人專注關鍵決策

降低維運複雜度與重複稽核負擔



AI 產出

加速開發



平台自動把關

測試・資安・合規



人做關鍵決策

風險判斷・例外核准



HITL：從靠人記得，變成可執行、可追蹤

平台承接重複檢查，工程師保留責任與判斷。

1

必然性多工？



参考資料

敏捷時代的五個拆解層次

階段	核心問題	主導角色	主要產出
需求拆解	為什麼需要？ 有哪些假設？	PO 主導，團隊共同探索	業務意圖、需求假設、業務情境、業務規則、驗收條件
設計拆解	系統如何承擔責任？	工程師主導，PO 參與確認	領域模型、設計決策、系統責任
工作拆解	如何可靠地完成？	工程師主導	Tasks、依賴關係、完成條件
增量整合	系統是否正確運作？	工程師主導，團隊共同驗證	Working Increment、測試結果
回饋調適	是否真正創造價值？	PO 與團隊共同主導	學習成果、Backlog 調整

如何檢測AI產出的功能程式是否正確?



Read the system：從模式開始

▶ 傳統理解系統

- 採取由下而上的方式：先看程式碼，再逐步理解整體結構
  - **Code → Function → Class → Module → System**

▶ AI agent 時代

- **System Behavior → Interaction Pattern → Responsibility → Implementation → Code**
  - 先回答：
    - **系統正在完成什麼事情？**
    - **這件事情遵循什麼行為模式？**
    - **哪些元件共同完成這個行為？**
    - **元件之間如何傳遞資訊與責任？**
    - **哪些程式碼實現了這些行為？**

如果明天出事了?!

- 我能說清楚它為什麼會這樣？

STEP 1｜ 先不要讓 AI Coding：先把需求改寫成「可驗證的假設」

STEP 2｜ **工作拆解**：不要再按照「程式結構」切工作

STEP 4｜ AI 產出之後，不要立刻 Code Review

STEP 5｜ Code Review 改成「**模型 Review**」

STEP 6｜ 從 Code Review 進入「理解 Review」 - 五件事

STEP 7｜ 把「我覺得懂」變成可觀察的理解程度

(因此 Merge 前最後做一次：Responsibility Check)

Code Review

AI 時代的工程師

理解的五個層次

從知道改動，到預測行為

理解逐步深化

第五層 我可以預測修改後的行為？



第四層 我知道失敗時去哪裡找證據？



第三層 我知道哪裡可能失敗？



第二層 我可以解釋運作的結構？



第一層 我知道 AI 改了甚麼？



理解愈深，愈能解釋、查證與預測。

由個人理解到共同知識

將理解缺口回饋到模型，形成持續演進的 Living Documenitation

1 **WHY**
為何要改？
需求模型
Impact Mapping

2 **HOW**
如何運作？
系統架構圖
C4 / 資料流程

3 **WHAT**
改了什麼？
行為模型
BDD / 狀態轉換

4 **WHAT IF**
可能出什麼問題？
風險模型
故障情境 / 邊界分析

5 **EVIDENCE**
如何確認是真的？
測試模型
可觀測性 / Log / Trace

AI
產出變更

- Code
- 說明文件
- 測試案例
- 架構圖
- ...

Code Review：五層理解檢驗（個人）

WHY 動機	HOW 結構	WHAT 變更	WHAT IF 風險	EVIDENCE 證據
我知道為何要改嗎？	我知道如何運作嗎？	我知道改了什麼嗎？	我知道哪裡會出問題嗎？	我有足夠證據相信嗎？

發現理解缺口或新的問題

理解缺口
回饋到模型

更新與補強（個人 → 團隊）

- 向 AI 提問補充說明
- 補充文件與範例
- 新增測試與驗證
- 更新架構與決策紀錄
- 整理為可重複使用的知識

理解足夠

團隊共同知識

被保存、共享、持續演進

Living Documentation

- 需求 (Why)
- 架構 (How)
- 行為規格 (What)
- 風險與決策 (What If)
- 測試與觀測 (Evidence)
- 經驗與範例 (Know-how)

持續演進
更少重複問題
更快的理解
更高的系統可信度

持續回饋，深化理解

AI 時代的工程師工作 SOP

AI 幫你做，但你要懂、要驗證、要負責

更快的產出，更可靠的工程

1 需求 → 假設

先釐清問題
建立可驗證的假設



回答四個問題

- 問題是什麼？
- 我認為原因是什麼？
- 我要看到什麼證據？
- 怎樣才算完成？

避免做一件
不該做的事

2 拆解工作

按可驗證的假設
拆分最小工作



每個工作單位包含

- Goal（目標）
- Change（改動）
- Evidence（證據）
- Boundary（邊界）

從技術元件拆
改為從驗證目標拆

3 AI Coding

在明確目標與限制下
讓 AI 產生方案



提供 AI 的關鍵資訊

- Context（背景）
- Goal（目標）
- Constraints（限制）
- Evidence（驗證方式）
- Output（輸出要求）

不是「幫我寫」
而是「在邊界裡產生方案」

4 產出說明

先理解改變與行為
再進入 Code Review



要求 AI 先提供

- Before（原本行為）
- Change（改了什麼）
- After（新的行為）
- Risk（可能風險）
- Evidence（如何驗證）

避免直接逐行看 Code
先建立全貌

5 Code Review

用模型化思維
進行完整審查



五個核心問題

- Why（為什麼要改？）
- What（改了什麼？）
- How（怎麼做到？）
- What if（如果失敗？）
- Evidence（如何證明？）

不是逐行讀
而是檢查系統行為與風險

6 理解檢查

確認自己到底
理解了多少



關掉 Code，回答五件事

- 我知道改了什麼？
- 我可以解釋怎麼運作？
- 我知道可能在哪裡失敗？
- 我知道去哪裡找證據？
- 我可以預測後的行為？

能不看 Code 解釋
才算真正理解

7 責任確認

確認風險、驗證方式
與應對路徑後再上線



合併前的四個檢查

- 我能解釋嗎？
- 我能驗證嗎？
- 我知道風險嗎？
- 出事時我知道
  去哪裡看嗎？

這是我按下 Merge，
我必須負責

持續迭代

發現新問題
→
回到前面的步驟

工程師的理解程度五層

L1	L2	L3	L4	L5
我知道 改了什麼	我可以解釋 怎麼運作	我知道可能 在哪裡失敗	我知道失敗時 去哪裡找證據	我可以預測 修改後的行為

四個 Yes 才能 Merge

- ☑ 我能解釋
- ☑ 我能驗證
- ☑ 我知道風險
- ☑ 我知道去哪裡看



AI 可以替我產生 Code，但不能替我產生理解。

Better Engineers × Better Systems × A More Reliable Tomorrow



問題 與 回答

歡迎提問，我們依流程進行回應



問題 10

分析 3

回答 1

確認 3

完成

High

Low

請寫下你想問的問題
並貼在問題區

問題描述

加以重點
描述

解答

在一分鐘內
做解答

點頭

提問者
確認解答

最後的
回顧檢視

AI 治理的本質：治理什麼？

概念	核心問題	治理對象
AI Governance	AI 應該如何被負責任地使用？	組織、政策、責任與風險
AI Engineering Governance	AI 如何被安全地整合進工程活動？	開發流程、驗證、部署與維護
AI Agent Governance	具有自主行動能力的 Agent 如何受到適當控制？	Agent 的權限、行為、工具與執行環境

從控制導向治理，轉向風險導向、回饋驅動的治理

Platform Engineering：AI 治理真正落地的關鍵

平台能力	治理目的	實際做法
Agent Registry	知道有哪些 Agent	登錄用途、負責人、版本及風險
Identity & Access	控制 Agent 能做什麼	最小權限、工具授權、短期憑證
Observability	知道 Agent 做了什麼	行動軌跡、工具呼叫、錯誤紀錄
Evaluation	知道 Agent 做得對不對	測試資料集、行為評估、回歸測試
Lifecycle Management	管理 Agent 的生命週期	發布、監控、停用、版本管理

軟體工程 3.0 構想

HILP

人類(**ACE**) → 寫 BriefingScript + LoopScript + MentorScript

↓

Agent(**AEE**) → 執行，必要時產生 **CRP**（請教人類）

↓

人類 → 用 **VCR** 回應 **CRP**

↓

Agent → 完成後產生 **MRP**（證明可合併）

↓

人類 → 用 **VCR** 審核 **MRP**（批准 / 要求修改）

> 用版本控制的結構化產物當「契約 / API」，讓人類與 agent 透過明確介面協作，而不是自由聊天。

ACE, Agent Command Environment

AEE, Agent Execution Environment

CRP 諮詢請求包 Consultation Request Pack

MRP合併就緒包 Merge-Readiness Pack,

VCR 版本控制決議 Version Controlled Resolutions

軟體工程 3.0 構想

HILP

1 人類(ACE)



寫

- BriefingScript
- LoopScript
- MentorScript

產出：三種 Script

2 Agent(AEE)



執行

必要時產生
CRP（請求人類）

產出：CRP

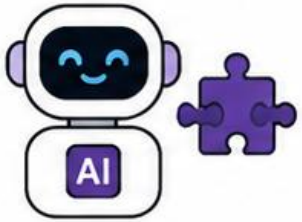
3 人類



用 VCR 回應 CRP

產出：VCR

4 Agent



完成後產生
MRP（證明可合併）

產出：MRP

5 人類



用 VCR 審核 MRP
（批准 / 要求修改）

產出：審核結果

用版本控制的結構化產物當「契約/API」，
讓人類與 agent 透過明確介面協作，
而不是自由聊天。

- ACE — Agent Command Environment 人類指揮環境
- AEE — Agent Execution Environment Agent 執行環境
- CRP — 諮詢請求包 Consultation Request Pack
- MRP — 合併就緒包 Merge-Readiness Pack
- VCR — 版本控制決議 Version Controlled Resolutions

介面明確 | 可追蹤 | 可合併 | 可控管 | 可擴展

從人工到完全自主的軟體工程層級

Ahmed E. Hassan
艾哈邁德·E·哈桑

- **Level 0：Manual Coding**(No-AI SE) **[SE 1.0]**
  典型用途：沒有 AI 映射；人類親自把想法轉成程式 token。
- **Level 1：Token Assistance**(AI-Augmented Coding) **[SE 1.5]**
  把開發者當下的編輯意圖映射為預測 token，例如現代 IDE 的自動完成。
- **Level 2：Task-Agentic**(AI-Augmented SE) **[SE 2.0]**
  把已規劃的程式修改，例如函式描述轉為完整程式區塊；測試與 code review 亦可採類似自動化。
- **Level 3：Goal-Agentic**(Agentic SE) **[SE 3.0]**
  把技術目標，例如「加入 caching layer」，映射成詳細的程式修改計畫。
- **Level 4：Specialized Domain Autonomy** **[SE 4.0]**
  把某個特定領域的廣泛技術使命。
- **Level 5：General Domain Autonomy** **[SE 5.0]**
  把一般性技術使命，例如「確保所有系統都很健壯」，轉化為任何陌生領域所需的特定技術使命。

需求拆解與工作拆解

PO 主導價值探索，工程師參與理解。

工程師主導技術探索，團隊共同辨識與挑戰假設。

雙方共同完成需求拆解與假設驗證。

需求拆解與工作拆解

比較項目	需求拆解	工作拆解
核心目的	理解問題	組織解決方案
拆解對象	業務意圖與行為	設計與實作工作
思考方向	從模糊到清楚	從複雜到可執行
主要問題	為什麼？需要什麼？	怎麼做？由誰做？
主要活動	提問、探索、假設、驗證	分工、排序、估算、執行
主要產出	業務規則、情境、驗收條件	Tasks、依賴關係、交付成果
成功標準	需求可被共同理解與驗證	工作可被執行與驗收
AI 的角色	協助提問、找出歧義	協助規劃、實作與測試
常見錯誤	太早決定解決方案	把工作拆得太細或彼此割裂

需求拆解的品質，取決於我們發現了多少重要的未知。

工作拆解的品質，取決於我們能否控制工作的複雜度與整合風險。

需求拆解：理解意圖，釐清價值

原始需求	拆解後的需求
會員快到期時通知	一般會員到期前 7 天通知
VIP 提早通知	VIP 到期前 30 天通知
自動通知	系統每日檢查符合條件的會員
通知會員	透過會員允許的通知管道發送
避免干擾會員	已續約會員不再收到提醒
確認成效	衡量通知後的續約轉換率

透過提問、假設與驗證，逐步釐清使用者需求，形成可理解、可驗證的業務行為與規則。

提問

工作拆解四原則

四個提問，讓拆解有依據、可執行、可驗證

布魯姆提問法



01 對成果

這件事讓哪個需求往前一步？

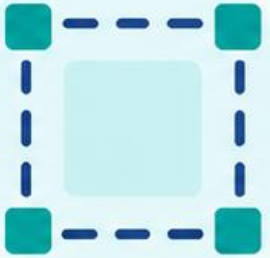
說清用途，連結需求



02 切邊界

負責到哪裡？怎樣算完成？

界定範圍，明確責任



03 查未知

哪些假設需要先確認？

找出未知，優先驗證



04 留證據

如何確認它做對了？

留下證據，共同檢查



成果對得上 · 邊界說得清 · 未知先確認 · 完成有證據

工作拆解：組織實作，降低複雜度

- 工作拆解是將已經釐清的需求與設計，分解成具有明確產出、依賴關係與完成條件的可執行工作單位。

主要回答：

- **How**： 如何實現需求？
- **Who / What**： 由誰或哪個 AI Agent 執行？
- **When**： 工作之間有什麼順序與依賴？
- **Done**： 如何確認工作已經完成？

工作拆解不應該只依照程式模組來切割，也應該考慮能否形成可驗證的工作成果。

需求拆解與工作拆解

- **需求拆解：解「為什麼、真正要什麼？」**
- **工作拆解：解「為了做到它，要做哪些事？」**

	需求拆解	工作拆解
核心問題	客戶真正想要什麼？	系統要做哪些事情？
對象	Business Intent	Feature / Function
關注	Why / Who / What	How / Component / Task
找的是	意圖、規則、假設、邊界	功能、責任、依賴、順序
主要產物	Requirement Model	Task Model
AI 的作用	**協助發現模糊與未知**	**協助拆成可執行 Task**
人的責任	確認意圖是否正確	確認拆法是否合理

軟體開發的核心議題

價值、實現、可信、學習，缺一不可



價值

方向不清，越快越可能做錯

先確認：為誰解決什麼問題？



實現

改不動，學到了也難以改善

讓想法能落地，讓系統能持續修改。



可信

缺乏證據，就難以判斷對錯

用測試與運行證據，支持每一次改動。



學習

沒有回饋，誤解就可能長期存在

從真實結果修正理解，調整下一步。

持續把理解，轉化為可信且有價值的軟體

軟體開發的核心議題

工程師的核心工作，是持續深化對專案的理解
並將其轉化為值得信任、能創造價值的軟體

面向	核心問題	可觀察的證據
價值	為誰解決什麼問題？ 為何值得做？	目標、優先順序、成功條件
實現	能否把理解轉為 可運行、可修改的系統？	從想法到可驗證成果的阻力與時間
可信	憑什麼相信系統符合預期， 且風險可接受？	驗收範例、測試、實際運行證據
學習	結果是否支持原先的理解？ 下一步該怎麼改？	從回饋到調整、再驗證的速度

AI SDLC：工程師需要理解什麼？

持續深化對專案的理解，將其轉化為可信且有價值的軟體

層面	需要理解什麼	理解的具體內容
平台基礎 脈絡與衡量	理解資訊依據與判斷限制	來源・時效・適用範圍｜指標定義・盲點
規劃	理解問題、意圖與限制	使用情境・領域規則・範圍・未知
設計	理解方案、相依關係與取捨	系統邊界・資料流・替代方案・失敗情境
開發	理解改動、影響與 Agent 工作狀態	程式結構・改動機制・任務邊界・進度
審查	理解證據、盲點與剩餘風險	驗證依據・測試涵蓋・未驗證事項
維護	理解運作、原因與演變歷史	真實行為・異常影響・因果・修復後果

關鍵能力：連接不同層面的理解，辨識矛盾，持續修正

如何檢驗 AI 產出的功能程式

五個關卡，讓驗收有依據

01

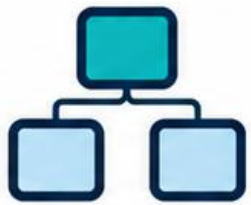


功能正確

輸入輸出對嗎？

正常・邊界・異常測試

02



容易維護

有隱藏耦合嗎？

檢查依賴與變更影響

03



符合需求

解決真正問題嗎？

對照需求與驗收條件

04



證據充分

憑什麼判定通過？

檢視測試結果與覆蓋

05



風險可控

出錯代價能承擔嗎？

確認責任與回復方案



驗收標準：證據足夠、風險可接受、未驗證假設已標明

依風險決定驗證深度，由人做最終判斷