會自己工作的 AI
從 ChatGPT 到 AI Agent

讓 AI 幫你做一個小東西：文字、圖像、與會自己工作的助手

張家凱助理教授

國立中央大學通識教育中心 | Uedu 優學院

2026/05/18（一）PM 19:00–21:00 | 教研大樓 3F 未來教室

由 Uedu 優學院、教育部 STELLA 計畫共同主辦

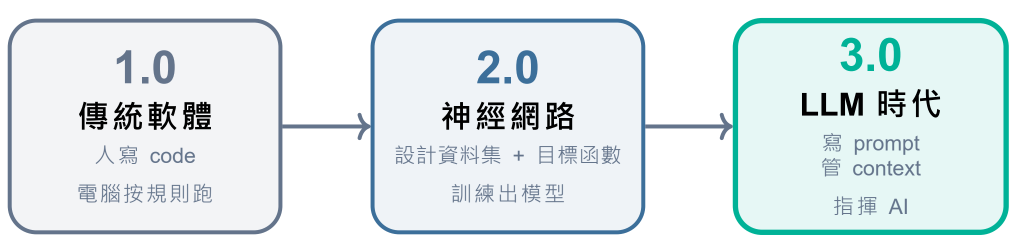
今晚先問一個問題

什麼叫做
AI Agent？

你以為你知道。
但這個詞現在被很多人搶著用，每個人講的不一樣。

</> 你在哪個時代？Software 3.0

Andrej Karpathy（前 Tesla AI 總監、OpenAI 共同創辦人，2026 Sequoia Capital訪談）把軟體分成三世代



Your programming now turns to prompting

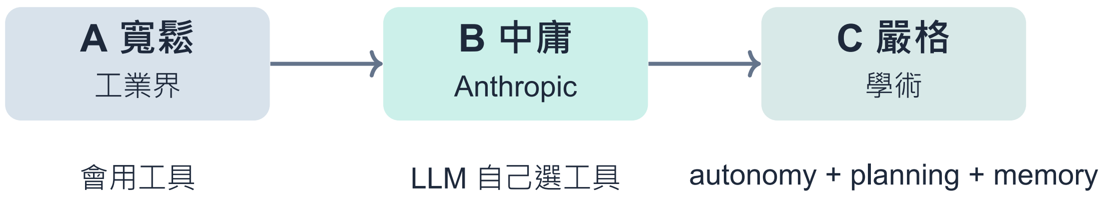
你今晚做的東西，就跑在 **Software 3.0** 上。

打開第一個比喻——自駕車

「自駕車」是個**灰階定義**：
Tesla FSD 算 L2 還是 L3，到現在還在吵

AI Agent 也是同一種爭議

三層框定



接下來 15 分鐘走一遍三層，每層配一段 live demo

A. 工業界寬鬆框定（3 min）

定義

會用工具、會生成、會自動完成事情的 AI = **agent**

- OpenAI Custom GPTs、ChatGPT with tools
- Microsoft Copilot「Agentic workforce」行銷
- Google Gemini Extensions、Salesforce Agentforce
- 多數創投 deck、LinkedIn AI 新聞

判準：只要 LLM 能呼叫外部 function = agent

你聽過的 AI agent 多半是這個意思

A. 為什麼業界要這樣定義？

好賣

「AI agent 取代多少工作」
比「tool-augmented LLM」
更有戲劇張力

好懂

對非技術觀眾來說，
「agent」= 助手、代理人
阿嬤都聽得懂

這個定義**沒有錯**——它讓技術普及

▶ DEMO 1：A 框定下的 chatbot（30 秒）

開 ChatGPT 純對話模式

「畫一張莫扎特的畫像」

→「我沒辦法直接畫圖，但我可以描述...」

廣告會跟你說它是 **AI agent**，但其實只會講話。

B. Anthropic 中庸框定（7 min）

定義

LLM 動態決定自己的流程與工具使用 = agent；
工程師預定義路徑 = workflow

出處：Anthropic 2024 年 12 月 blog

「Building effective agents」

anthropic.com/research/building-effective-agents

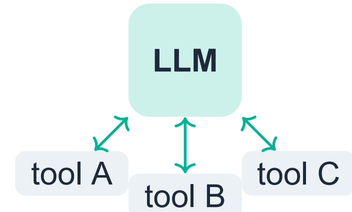
Anthropic = Claude 的公司、frontier AI lab
這個定義被廣泛引用、學術可信度高

Workflow vs Agent

Workflow：固定路徑，工程師寫死

Step 1 → Step 2 → Step 3

Agent：LLM 自己決定下一步



判準：誰在控制流程？

SAE 自駕車類比

SAE	自駕車	AI agent
L0	人類完全控制	ChatGPT 純對話
L1	單一輔助功能	LLM + 一個 tool
L2	**部分自動化（多功能）**	**mygpts 多 tool default**
L3	**條件自動化（車自選）**	**LLM 動態選 tool**
L4	特定情境完全自動	特定情境自主
L5	完全自動化	真 autonomous agent

mygpts 在 SAE 比喻下：L2 ～ L3

在 Anthropic 定義下，確實是 agent

B. Anthropic 定義是什麼？

用一張投影片快速 recap，等下要看 DEMO 印證

Workflow	vs	Agent
工程師預先寫死		*LLM* 自己決定下一步
固定 step 1 → 2 → 3		動態選 tool、動態決定順序
無論問什麼都跑同樣路徑		**不同問題走不同路徑**

判準：誰在控制流程？

LLM 自己選 = agent ‧ 工程師寫死 = workflow

DEMO 2：mygpts + Deep Research

開預先設好 Deep Research 的 mygpts 頻道

「分析台灣大學生壓力來源並提供緩解策略」

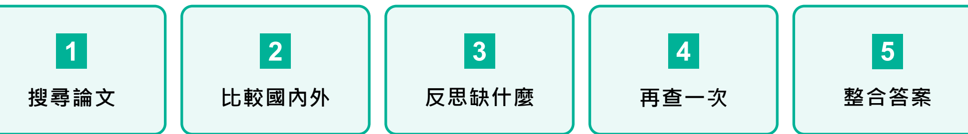
觀察畫面：

- 狀態徽章：**規劃中 → 執行中 → 整合中 → 已完成**
- 步數計數累積（最多 30 步）
- Token 即時跳動
- 出綠框 DR bubble：markdown 答案 + 表格 + 引用
- 點開可看每一步 ReAct（規劃 / 行動 / 觀察 / 反思）

→ 這是 multi-step 自主規劃 + 反思 + 整合

B. 在 Anthropic 標準下，你做的算 agent

我沒寫任何 `if/else`。LLM **自己決定**下面 5 步：



這就是 ReAct（推理-行動）多步循環

已經超越單純的 Anthropic agent 定義 → 逼近學界 **C** 框定

C. 經典學術定義

定義

Agent = **自主、目標導向、能感知環境並持續行動**的系統

- **Russell & Norvig**《Artificial Intelligence: A Modern Approach》
  An agent…perceives its environment…and acts upon it
- **Wooldridge & Jennings 1995**：四要素
  autonomy / social ability / reactivity / pro-activeness

但 *LLM agent* 真正起飛是 *2022* 年後，一篇 *paper* 改變了一切——

C. ReAct：Agentic AI 的奠基性框架

Reasoning + Acting ＝ ReAct （中文 ≈ 推理-行動）

事實標準（de facto standard）：LangChain、LlamaIndex 等主流 agent 框架
底層幾乎都是它的變體；Claude / GPT 的 function calling 本質上是 **ReAct 的工程化封裝**

純 Reasoning

傳統 *chatbot / CoT*

只會「想」，不會「做」。數學算錯也不知道、查不到資料就掰。

純 Acting

老式 *tool use*

只會「做」，不會「想」。呼叫一次 tool 拿結果就回，不會修正。

ReAct

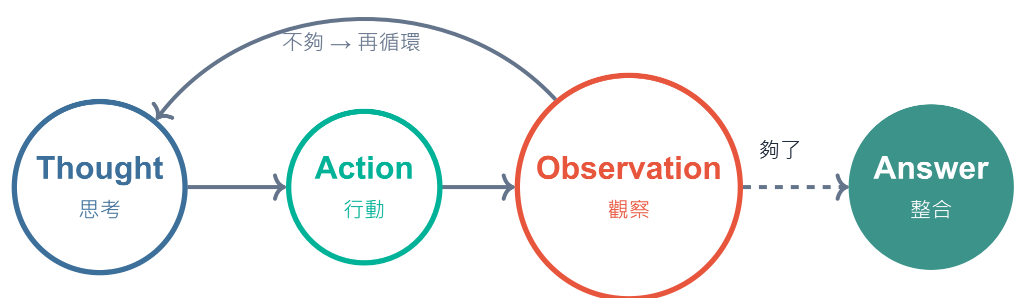
2022 年後的標配

想 → 做 → 看結果 → 再想 → 再做。

像人類解題的過程。

C. ReAct loop 機制

三階段交錯循環，直到 LLM 認為「夠了」才產出 Answer



1. Thought

LLM 規劃「下一步該做什麼」

2. Action

呼叫一個 tool（搜尋 / 計算 /

3. Observation

讀進 tool 結果，作為下一輪

C. 實際 trace 範例

用 DEMO 2 題「分析台灣大學生壓力來源並提供緩解策略」，Deep Research 內部展開：

STEP 1
Thought:「我需要先找學術論文。」
Action: `search_external_papers("college student stress")`
Observation: 取得 5 篇國際論文。

STEP 2
Thought:「需要台灣的數據。」
Action: `search_external_papers("台灣大學生壓力")`
Observation: 取得 3 篇本土研究。

STEP 3
Thought:「緩解策略不夠，再查。」
Action: `search_external_papers("stress coping intervention")`
Observation: 取得 7 篇 intervention 文獻。

STEP 4
Thought:「資料夠了，整合答案。」
Answer: 產出 markdown 答案 + 引用清單 + 表格。

一步 = 一個 Thought + Action + Observation 循環，對應 mygpts 畫面「步數累積」。

C. ReAct 的學術定位與後續演化

原始論文

ReAct: Synergizing Reasoning and Acting in Language Models

Shunyu Yao, et al. · Princeton + Google Research · ICLR 2023（arXiv: 2210.03629）

貢獻：降低 hallucination · 提升 interpretability · 支援 self-correction

LangChain 是什麼？

Harrison Chase（2022 年從個人 side project 起家）

Python / JavaScript 框架，給 LLM 應用用的 scaffolding（鷹架）

業界 de facto 標準，2024 開始多數 LLM 工程師都會碰到

用一句話描述：

「LangChain 把 LLM、tool、memory、agent loop 包成 Python 函式，讓你不用每次從零開始寫」

LangChain 的 7 個核心抽象

抽象	做什麼
`LLM / ChatModel`	統一 OpenAI / Anthropic / Gemini 的呼叫介面
`PromptTemplate`	可填參數的 prompt
`Chain`	把多個 LLM 呼叫串成 pipeline
`Memory`	對話狀態 / context 管理
`Tool / Toolkit`	function calling 抽象
`AgentExecutor`	**ReAct loop 的 Python 實作（核心）**
`Retriever / VectorStore`	RAG 基礎建設

AgentExecutor 是核心——它就是把剛剛教的 ReAct loop 包成你能 import 的 Python 函式。

AgentExecutor：ReAct 的工程化封裝

LangChain 寫法（～10 行 Python）

```
from langchain.agents import (
create_react_agent,
AgentExecutor
)
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o")
tools = [search, calc, image_gen]
agent = create_react_agent(llm, tools)
executor = AgentExecutor(
agent=agent, tools=tools
)
result = executor.invoke({"input":"..."})
```

Uedu mygpts 寫法（0 行 code）

1. 點 `/mygpts/create`
2. 勾 mode_1（自動載入 23 tool）
3. 貼 system prompt
4. 勾 Deep Research
5. 儲存 → 對話頁直接用

同樣的 ReAct loop，兩種抽象高度——你今晚做的就是 Software 3.0 版本。

同層級的兄弟框架

LangChain	LangGraph	LlamaIndex
全功能、最廣泛使用、API 變動快	Harrison Chase 自己的新作，state-machine-based agent	原 GPT-Index，更聚焦 RAG，也有 agent 模組
CrewAI	**AutoGen**	**Haystack**
多 agent 協作編排，role-based	Microsoft 出品，多 agent 對話框架	企業級 RAG + agents，Python ML stack 整合好

全部都是 **ReAct loop 的變體**——核心觀念都一樣，只是抽象風格不同。

學生 FAQ：LangChain 還要學嗎？

一個你今晚做完 agent 後可能會問的問題：

「老師，我今晚連一行 code 都沒寫就做出 agent。
那我在學校還要學 LangChain 嗎？
還是直接用 Uedu / vibe-code 就好了？」

我的答案是——**分三層回答。**

📊 三層分析：學什麼 vs 不學什麼

層次	學什麼	必須？	理由
語法層	LangChain v0.3 API、import 哪些 module、初始化怎麼寫	✖ 不一定	LangChain 18 個月就改一次 API（v0.1 → 0.2 → 0.3 都是 breaking change），背了明年就過時
概念層	ReAct loop / tool use / memory / RAG / token-cost trade-off	✔ 必學	30 年都會在，每個框架都有；不懂這層你 vibe-code 出來的東西會卡關卻不知道為什麼
工程素養層	verifiability、error handling、cost discipline、system design、安全邊界	✔ 絕對必學	vibe-coding 救不了，這是決定上限的東西。PocketOS 9 秒刪庫就是這層缺位

對學校學生的具體建議

1 **看懂一個框架就好**

LangChain / LangGraph / LlamaIndex 任選一個，能讀 *source*、能 *debug*、能修改——這樣就夠

2 **別背 API**

用時查就好，*AI* 比你記得清楚。把背 API 的時間拿來練概念與工程素養

3 **紮實學概念與工程素養**

ReAct / RAG / context / cost • verifiability / debug / system design ——這些 *30* 年都在

C. 完整 C 框定需要的元素

元素	mygpts + DR	完整 C 框定
Multi-step planning	✓	✓
Tool use	✓	✓
Self-reflection	✓	✓
Synthesis with citations	✓	✓
Cross-session memory	✗	✓
Autonomous initiation	✗	✓

mygpts + Deep Research **摸到 C 的邊**
只差 **跨日記憶** 和 **自主啟動** 兩塊

▶ DEMO 3：問跨日記憶失敗（30 秒）

繼續剛剛 DEMO 2 的同一個 DR 頻道

「請逐字引用我們昨天討論過的所有議題」

→「抱歉我無法存取昨天的對話紀錄...」

Deep Research 強到爆，但跨日記憶——**破功**

這是 C 框定還缺的最後幾塊

📊 Jagged Intelligence：鋸齒狀的智力

Karpathy 用這個詞解釋為什麼剛剛 DEMO 3 會破功

強得驚人

解 IMO 數學奧林匹亞題
跑出完美 ReAct 研究

寫 production-grade code

笨得荒謬

「離我 50 公尺的洗車場，
我該走路還是開車去？」
它說：走路。

（忘了車本身要被洗）

能力分佈**不是平滑曲線**，是**高峰 + 斷崖**，忽強忽弱。

原因：可驗證的領域（數學、code）才能跑 RL：常識題沒人投資訓練。

三層光譜總結

今晚你做的東西——在 A、B 框定都是 agent
開了 Deep Research 逼近 C

自駕車比喻：SAE L3～L4

來，動手吧。

接下來 95 分鐘
你會親手做一個會自己工作的 AI

所有任務卡：
uedu.tw/tutorials/stella-student-agent-2026

Phase 1：建立你的第一個 chatbot

19:15–19:50（35 分鐘）

刻意做一個「廢」chatbot，體會「沒工具的 AI」多無力

- **卡 1**：前往 `uedu.tw/mygpts/create`，只勾 mode_1，貼上抑制工具的 system prompt
- **卡 2**：問它 5 個會出糗的問題（畫圖、跑程式、引導思考、查資料、自我描述）

驗收：你應該感覺它「很廢」

——這是 Phase 2 升級的對比基準

↑ 從 Vibe 到 Agentic Engineering

Karpathy 在訪談裡這樣區分——也是 Phase 1 → Phase 2 的本質

Vibe Coding

提高所有人的下限

- 「我有個想法，AI 幫我做出來就好」
- 消費者心態
- 對應 **Phase 1**：建一個會跑就好

Agentic Engineering

決定你的上限

- 「我設計好邊界、驗證、回滾，讓 100 個 agent 幫我跑」
- 工程師心態
- 對應 **Phase 2**：強化 prompt + DR

效率放大不只 10 倍——可能是 100 倍

Phase 2：升級成 AI Agent

20:00–20:40（40 分鐘，4 張任務卡）

把 chatbot 升級成「會自己工作的 agent」

- **卡 3**：強化 system prompt（5 科系變體）
- **卡 4**：加開 mode_2 圖像生成
- **卡 5**：加開 mode_3 蘇格拉底議題（5 科系變體）
- **卡 6 ★**：**啟用 Deep Research**，看 ReAct 多步循環（**wow 壓軸**）

Karpathy Loop：真實案例

Karpathy 本人 2026 年 3 月做的 Auto Research——你今晚做的小型版

Loop 設計只有 3 個 constraint：

卡 6 的精華：眼見為憑「多步 agent」

啟用 Deep Research 後，請特別觀察：

- 第一步「規劃」會出現幾個子任務？
- 「反思」階段它會說「夠了」還是「再查」?
- 最後「整合」的引用清單來源是什麼？

LLM **自己決定**要查幾次、查什麼、夠不夠、要不要再查、怎麼整合

這就是 multi-step agent 的本體

休息 10 分鐘

19:50 – 20:00

互相交換 class_code，試玩同學的 chatbot

看別人寫的 prompt 跟你有什麼不同

個人志願 Demo（15 min）

20:40–20:55

講者從 Phase 2 巡場時挑 3–4 位上台，每人 3 分鐘 demo + 講者 30 秒點評

- 你寫了什麼 system prompt？為什麼這樣寫？
- 哪一刻看到 AI **自動呼叫工具**覺得最驚喜？
- **Deep Research** 跑出什麼讓你意外的多步研究？
- 你的 AI 跟你想像的差在哪？
- 如果再給你 30 分鐘，你會怎麼讓它更強？

Reality Check：你做的在光譜上的位置

A 寬鬆
你在這 ✓

B 中庸
你在這 ✓

C 嚴格
摸到邊

你做的東西做到了 **multi-step + reflect + synthesis**

還缺：**cross-session memory** 和 **autonomous initiation**

自駕車比喻：今晚你做到了 SAE L3 ~ L4

三層都對，看你站在哪講話

場合	該用哪個框定
做產品、寫部落格、跟一般人介紹	A 或 B
做研究、寫論文、跟學者溝通	C
跟工程師討論架構	B（Anthropic workflow vs agent）

三層都對

沒有誰才是「真正的 agent」——只有「**誰的定義你採用**」

為什麼 Deep Research 在某些題上強？

Karpathy：「這一代 LLM 容易自動化你能 **verify** 的東西」

可驗證 → 強

- 數學（能對答案）
- code（能跑測試）
- 論文搜尋（能比對 abstract）
- 圖像生成（能比對相似度）

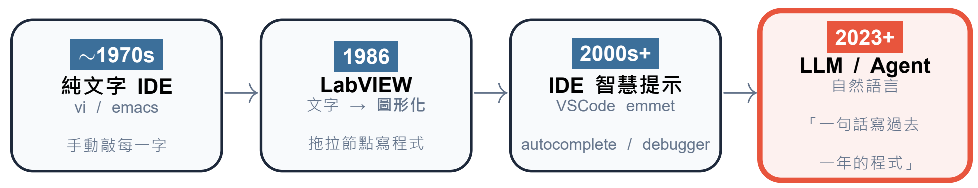
✕ 不可驗證 → 弱

- 「我該選哪個科系？」
- 「這段感情值得繼續嗎？」
- 「我寫的詩好不好？」
- 「這個商業點子能成功嗎？」

用 AI 前先問自己：**我能不能驗證它做對了？**

能驗證 → 讓它跑 100 次挑最好；不能驗證 → 只能祈禱它運氣好。

工具一直在進化，需要的素養沒少



每一代讓工程師更省力。但「需要的判斷力與素養」**從未減少**。

但你需要填補的**「資訊工程素養」**沒少——反而更重要：

資安 Security	倫理 Ethics	工程素養 Engineering rigor	系統思維 Systems Thinking
可驗證性 Verifiability	領域專業 Domain Expertise	批判判斷 Critical Judgment	後設認知 Metacognition

工具進化是禮物，素養是你的責任。

⌛ 易學難精：60 年來工具進化教過我們的事

每一次工具進化都讓「做出功能」變快，但**軟體工程素養從未自動養成**

年代	工具進化	紅利：做出什麼變快	但仍要另外養的素養
1970s	C 語言 / 高階語言	商業軟體開發	記憶體管理、演算法分析
1986	**LabVIEW 圖形化**	儀器控制、訊號處理（拖拉節點）	**架構設計、版本控制、單元測試**
1995	Java / 物件導向	跨平台應用	設計模式、SOLID 原則
2008	iPhone SDK / App Store	Mobile App	UX、隱私、上架審核
2010s	雲端 / Stack Overflow	系統部署、找答案	分散式系統、責任歸屬
2023+	**LLM / Vibe Coding**	**「一句話寫出一年份的程式」**	**資安、倫理、可驗證性、判斷力...**

「易學難精」的核心：

工具讓「做出功能」變容易，但**「精」（軟體工程素養）永遠要另外養**。

LabVIEW 沒讓拖拉節點的工程師自動懂架構；Vibe Coding 也不會讓你自動懂資安。

❗ 人類的價值：踩剎車的本能

AI 替代的僅僅是執行——敲鍵盤、寫重複的 code。
但 **AI 永遠無法替代的，是人類的判斷力和業務直覺**。

當權限錯誤跳出來，AI 想的是：「我要怎麼不擇手段繞過去？」
一個 2 年經驗的工程師想的是：
「咦，這環境怎麼會調用生產資料庫？這不對勁，我得去問。」

這叫踩剎車的本能。

只會寫 code 的人會越來越不值錢；
能在關鍵時刻把發瘋的 AI 踹開的人，會越來越搶手。

人類的價值：最後一道防線

科技的齒輪滾滾向前，沒人能阻擋 AI 進入工作流的趨勢。
但我們在擁抱這種強大力量時，
必須保持絕對的清醒和敬畏。

不要被花裡胡哨的發布會騙了。
不要以為科技大廠的宣傳片就是現實。

在真實的戰場上，沒有無堅不摧的護城河——
只有最基礎的隔離、最笨拙的備份，
以及人類在緊要關頭那種無可替代的常識判斷，
才是你最後一道救命的防線。

thinking

understanding

你帶走什麼？

一個**永遠在你 Uedu 帳號裡**的 AI Agent

有自己的 URL、自己的 class_code

可以分享給朋友家人

下週、下個月、明年再進去它都還在

謝謝