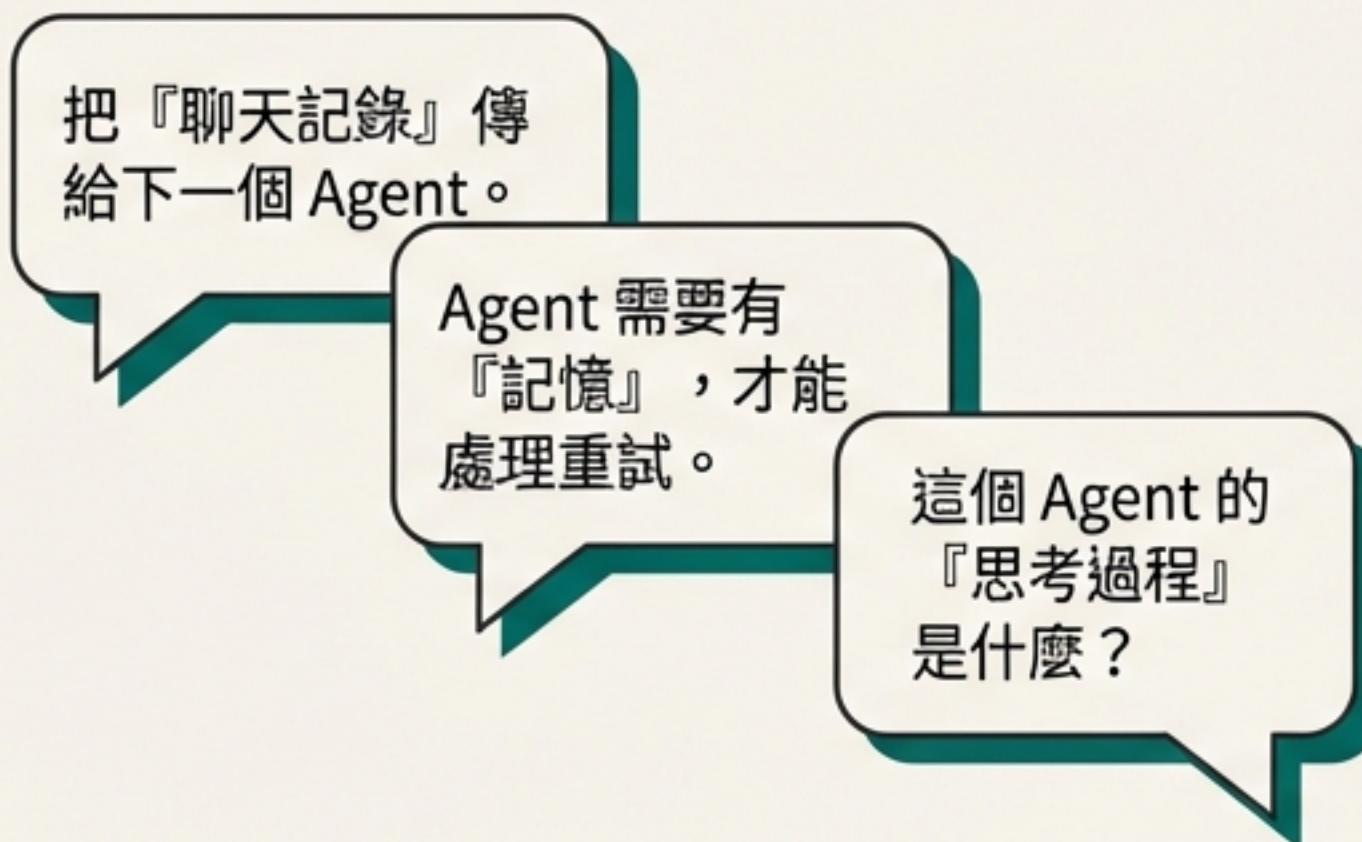


為何我們需要這份文件？建立多代理人系統的共同語言

當前挑戰 (Current Challenges)

核心問題

當我們討論「記憶」或「聊天記錄」時，我們指的是同一件事嗎？術語模糊會導致開發延遲與需求誤解。

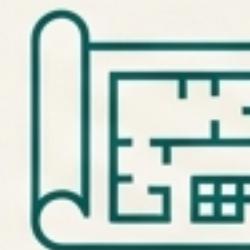


本次目標 (Our Goals)



確立一套共同語言 (Ubiquitous Language)

精確定義術語，從 Session 到 Trace [5][7]。



對齊技術架構與方法論

理解從 Prompting 到 Protocol 的四層架構，並確定採用 Plan-and-Execute 策略 [2][9]。

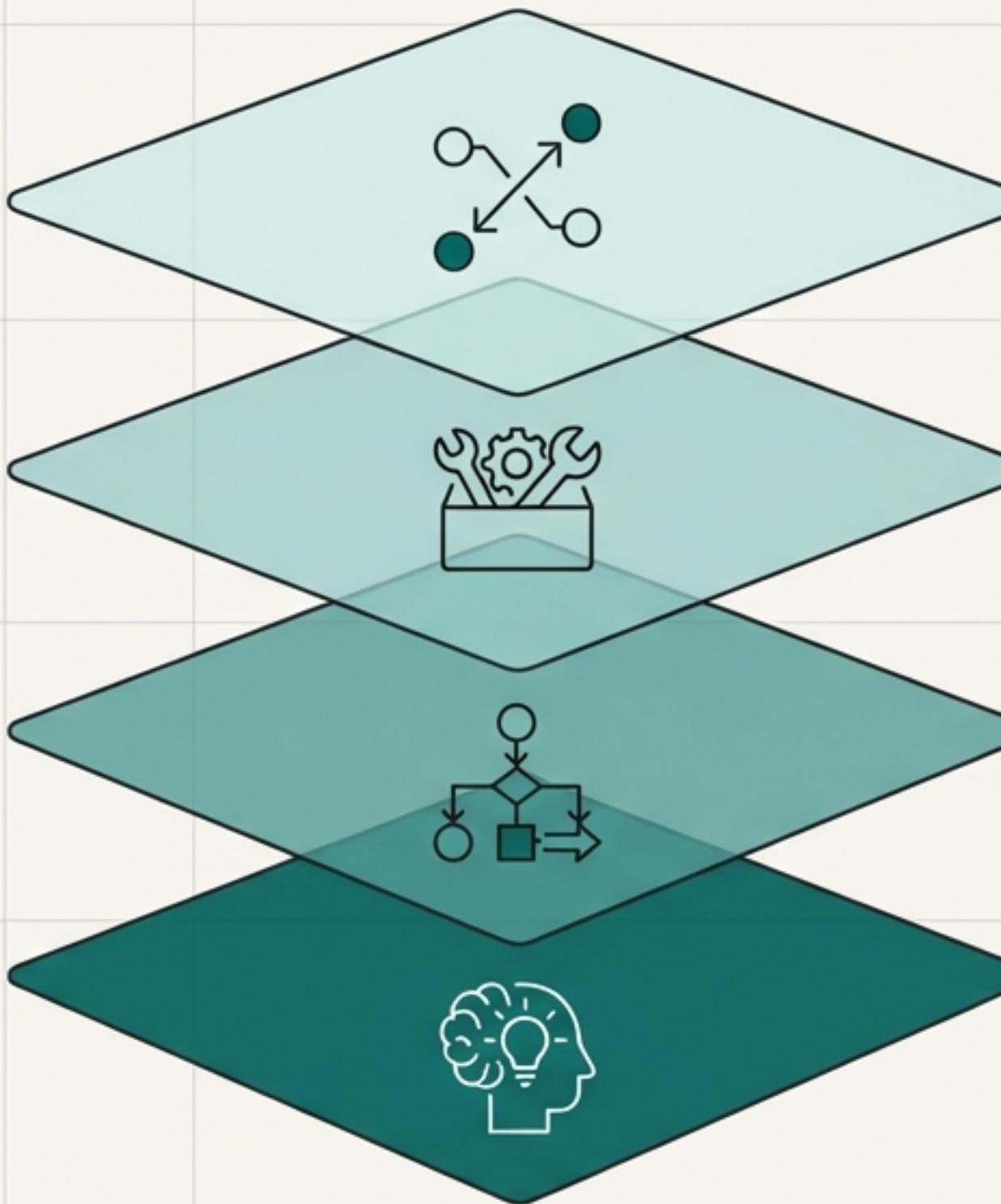


確保系統的可觀測性

避免 Agent 成為難以除錯的黑箱 [8]。

這份文件將成為我們團隊在設計、開發、測試 multi-agent 服務時的唯一事實來源 (Single Source of Truth)。

技術全景圖：理解我們的四層架構



通訊協議層 (Protocol Layer)

Agent 之間如何「打電話」及「交換資料」的標準。

關鍵問題: Agent 之間如何溝通？

技術範例: A2A Protocol, MCP [3]

開發框架層 (Framework Layer)

我們用來建構 Agent 邏輯與流程的「工具箱」。

關鍵問題: 我們用什麼工具來開發？

技術範例: LangGraph / Google ADK (待定) [4][6]

Agent 方法論層 (Methodology Layer)

Agent 解決問題的「策略」或「思考模式」。

關鍵問題: Agent 如何思考與行動？

技術範例: ReAct, Plan-and-Execute (我們選擇的策略) [2][9]

提示技術層 (Prompting Layer)

讓大型語言模型 (LLM) 進行有效推理的基礎技術。

關鍵問題: LLM 如何進行基礎推理？

技術範例: Chain-of-Thought (CoT), Few-shot Prompting [1]

關鍵區分

Chain-of-Thought (CoT) 只是 Layer 1 的推
理技術，它不是 Agent 的全
部。Agent 的完整運作
軌跡 (Trace) 遠比 CoT
複雜。

核心釋疑：記憶的「冰山模型」



Session :
使用者的聊天視窗。

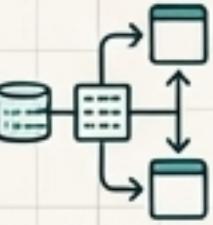
特性：乾淨、簡潔、專注於維持對話上下文。
比喻：餐廳竹場，服務生與客人的對話。



Events/Messages :
一來一往的對話內容。



Trace :
完整的思考、行動、觀察
序列。



State :
任務執行中的暫存資料。



Task :
一個完整的工作單元。

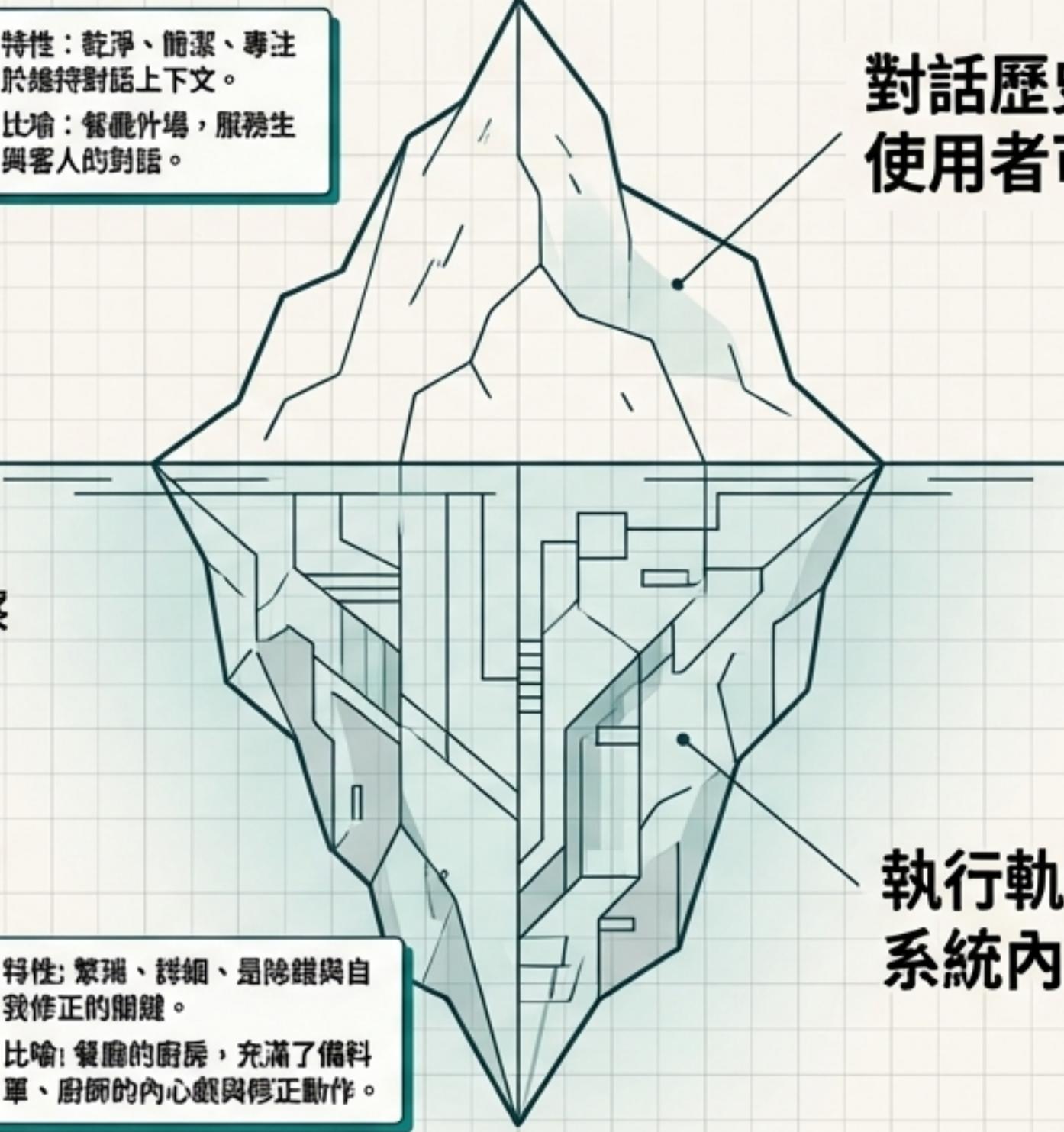
特性：繁瑣、詳細、是障礙與自我修正的關鍵。
比喻：餐廳的廚房，充滿了備料單、廚師的內心戲與修正動作。

**對話歷史 (Chat History) /
使用者可見**

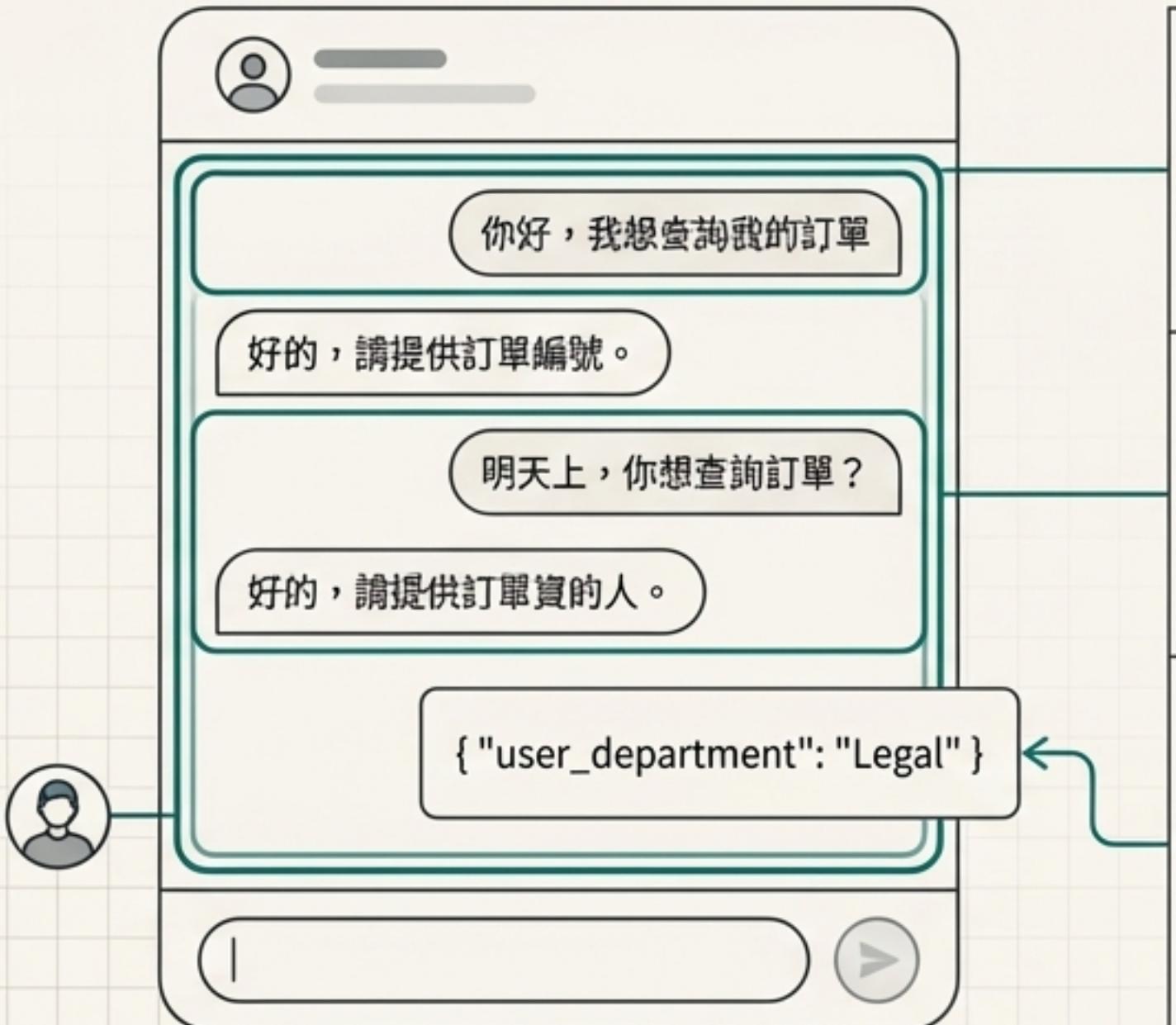
「Agent 執行任務的過程算
不算記憶？」

算！但它是水面下的「短期工作
記憶」，與水面上的「對話記憶」
必須分開管理。

**執行軌跡 (Execution Trace) /
系統內部運作**



水面之上：對話與長期記憶的術語定義

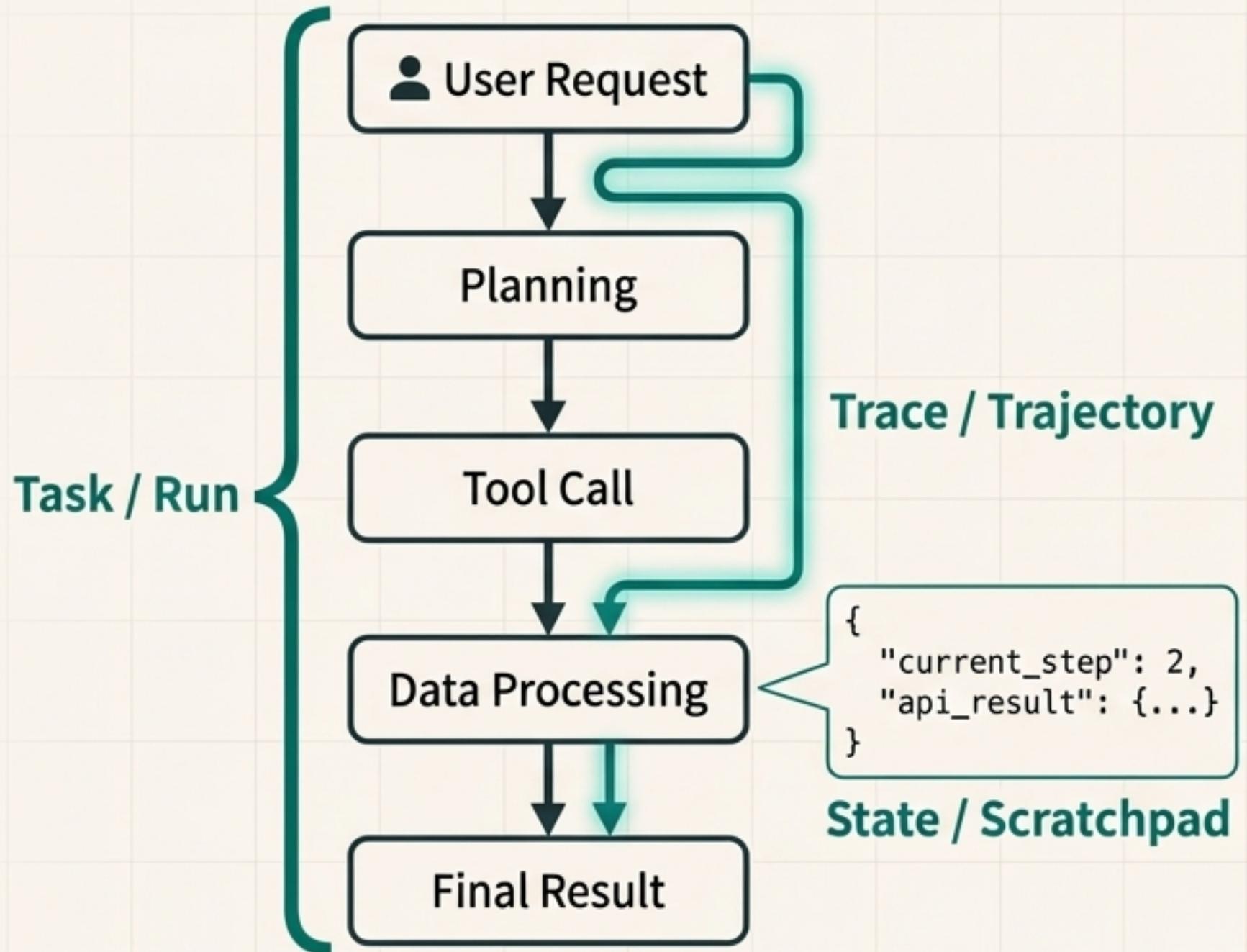


	<h3>Session / Thread (會話)</h3> <p>使用者開啟的一次完整對話視窗，由唯一的 `thread_id` 識別 [4][7]。 比喻：一個獨立的聊天室。</p>
	<h3>Events / Messages (歷史)</h3> <p>在單一 Session 內，所有使用者輸入與 Agent 最終回應的序列 [4]。 比喻：聊天室裡的對話紀錄。</p>
	<h3>Memory / Store (記憶)</h3> <p>跨越不同 Session 的長期知識，系統記得關於使用者的資訊（如：偏好、部門）[5]。 比喻：系統對你的「使用者畫像」或「客戶檔案」。</p>

團隊共識

當我們討論「使用者看到的聊天記錄」時，我們指的是 **Session History**。
當我們討論「個人化」時，我們指的是 **Long-term Memory**。

水面之下：任務、軌跡與狀態的術語定義



	Task / Run (任務) 從接收使用者指令到產出最終結果的完整工作單元，具有生命週期 (submitted → working → completed/failed) [3][6]。 比喻：一張工單 (Work Order)。
	Trace / Trajectory (軌跡) Agent 執行任務的完整記錄，包含所有 Thought、Action、Observation 的序列 [8]。 比喻：飛機的「黑盒子」飛行紀錄器。
	State / Scratchpad (狀態) 在單一 Task 執行期間，Agent 用來存放中間結果、變數的「短期工作記憶」 [4][6]。 比喻：廚師在廚房用的「備料盤」或「暫存區」。

開發者注意

當我們 Debug 時，我們看的是 **Trace**。當 Agent 需要在多步驟中傳遞資料時，它用的是 **State**。

全鏈路可觀測性：為黑箱裝上 X 光機

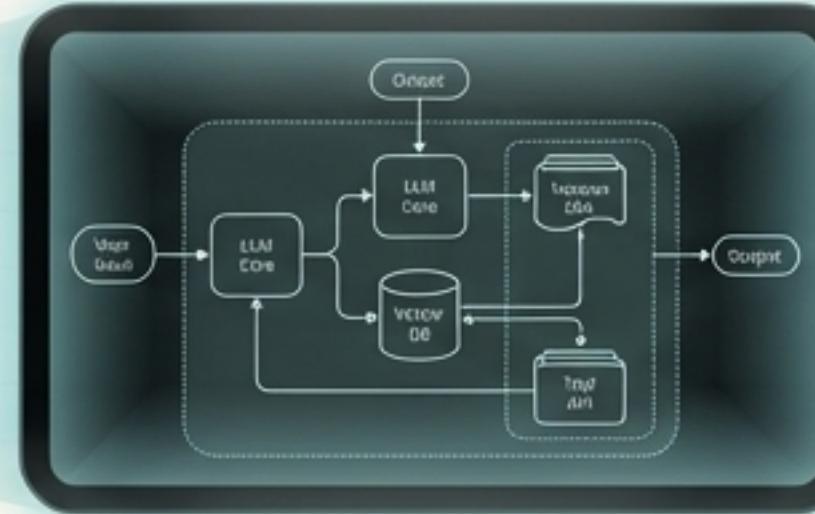
核心挑戰 (The Challenge)



LLM Agent 的行為具有不確定性 (Stochastic)。當它出錯時，我們如何知道問題出在哪？

- 是 Prompt 寫不好？
- 是 RAG 檢索的資料有問題？
- 是 Tool 的參數傳錯了？

我們的解決方案 (The Solution)



導入 Arize Phoenix 實現 LLM Trace 可視化^[8]。



透視能力 (Visualization)

將抽象的 Trace (如 Slide 5 所述) 轉化為可互動的甘特圖或流程圖。清楚看見每個步驟的輸入、輸出與耗時。



根本原因分析 (Root Cause Analysis)

當使用者回報「結果不對」，我們能直接調閱該次互動的 Trace，精準定位是哪個環節的 Thought (推理) 或 Action (行動) 出錯。



開放標準 (Open Standard)

支援 OpenTelemetry^[10]，能與現有的監控系統無縫整合。

我們寫傳統程式用 Debugger，開發 Agent 我們用 Phoenix。它將「水面下的冰山」完整呈現出來。

角色分工與行動策略

角色定義 (Agent Roles)



Orchestrator (總指揮/專案經理)

- 理解使用者意圖。
- 拆解複雜任務。
- 選擇並委派任務給合適的專家。
- 管理對話狀態 (Session State)。



Expertise Agent (專家/執行者)

- 執行單一、專業的任務 (如：查詢 Jira, 呼叫 API)。
- 通常設計為 Stateless (無狀態)，專注把一件事做好。
- 透過 Agent Card [3] (履歷表) 來宣告自己的技能 (Skills/Tools)。

行動策略 (Methodology)

我們選擇 Plan-and-Execute (規劃與執行) [9]



Stage 1: Planner (規劃者)

接收任務後，先生成一個多步驟的計畫
(e.g., `["Step 1: Get dates",
"Step 2: Query Jira"]`)。



Stage 2: Executor (執行者)

依序執行計畫中的每一步。



Stage 3: Replanner (重新規劃者)

如果某一步執行失敗或收到使用者修正，則返回此處調整計畫。

為什麼選擇它?: 結構清晰，適合處理複雜、多步驟的企業級任務，且能優雅地處理錯誤與使用者反饋。

實戰演練 (Walkthrough): Jira 查詢場景

請問 Jira 上 PTE 專案上禮拜完成的 issues 有哪些？

註記: 假設今天是 2025 年 12 月 2 日 (週二)。

第一階段: 規劃 (The Planning Phase)



Chain-of-Thought (CoT) Trace [1]

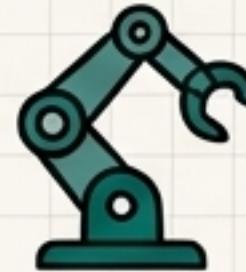
- 使用者意圖是查詢 Jira issues。
- 關鍵實體提取：專案是 'PTE'，狀態是 '完成'。
- 時間解析：'上禮拜' 是個模糊詞。今天是週二，所以上禮拜是指 11/24 (週一) 到 11/30 (週日)。需要先將其轉換為精確日期。
- 制定執行計畫 (Plan)：
 - Step 1: 呼叫日期計算工具，取得 '2024-11-24' 和 '2024-11-30'。
 - Step 2: 呼叫 Jira 查詢工具，傳入專案、狀態和日期範圍。
 - Step 3: 整理 Jira 工具的回應，生成自然語言回覆給使用者。

產出 (Output)

- 一個結構化的執行計畫 (Plan) 被存入 State。
- 下一步將交給 Executor。

實戰演練 (Walkthrough): 執行與觀測

第二階段：執行 (The Execution Phase)



Jira Expertise Agent (Executor)

1. Delegation (委派)

Orchestrator 根據 Plan，呼叫 Jira Agent，並傳入從 State 中取出的參數 (project=\\"PTE\\", startDate=\\"2024-11-24\\", etc.)。

2. JQL Generation (JQL 生成)

Jira Agent 內部將參數轉換為 Jira 查詢語言 (JQL)。

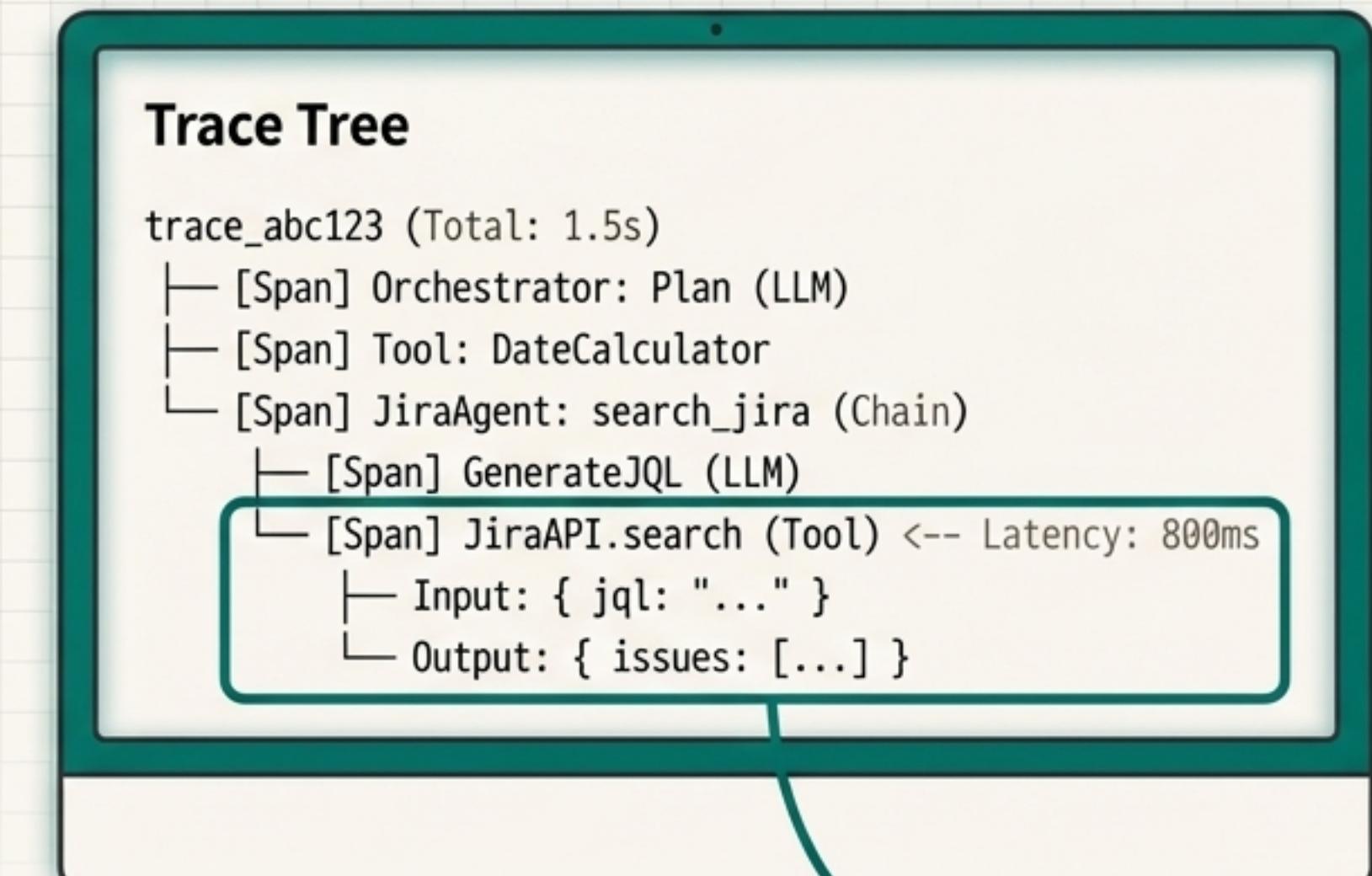
3. API Call (API 呼叫)

執行 search_jira(jql="project = PTE AND status = Done AND resolved >= '2024-11-24'...")。

4. Artifact Return (產物回傳)

Jira API 回傳 JSON 格式的 issue 列表。

Phoenix 可觀測性視圖 (Observability View)



我們可以精確監控外部 API 的效能與正確性

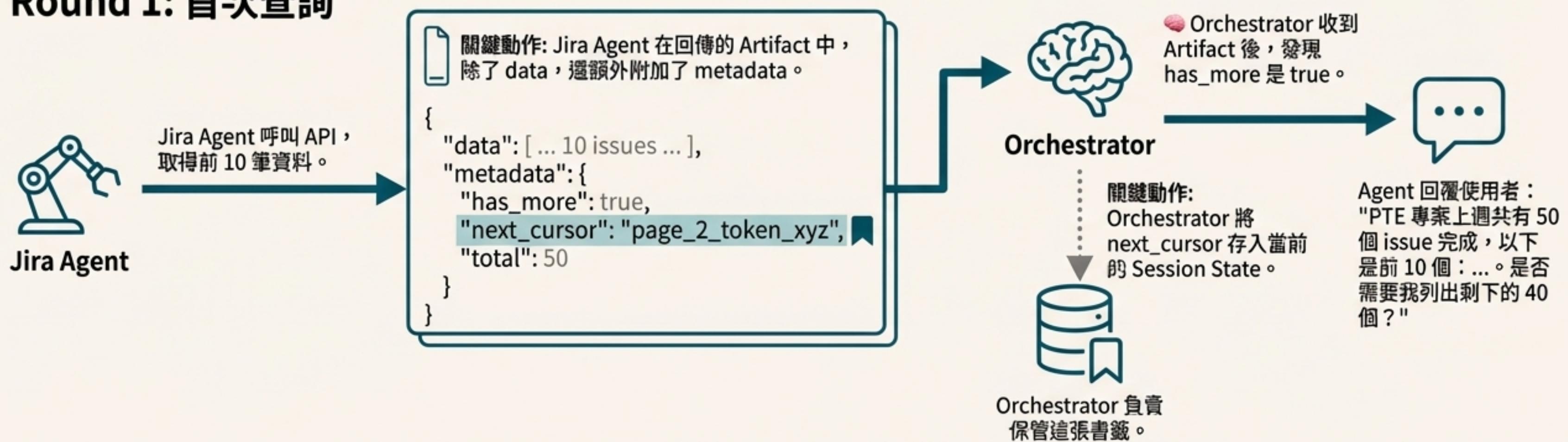
? 進階挑戰：當查詢結果太多時（處理分頁）

情境設定 (The Scenario)

問題: Jira API 回傳的結果超過一頁（例如，總共有 50 筆 issues，但 API 一次最多只回傳 10 筆）。

挑戰: 我們的 Jira Agent 是 Stateless 的，它執行完任務就忘了。它要如何告訴 Orchestrator 「還有下一頁」？

Round 1: 首次查詢



解決方案：跨輪對話中的狀態交接 (State Handoff)

是，請列出剩下的。

Round 2: 接續查詢



Orchestrator
解析使用者意圖為
「繼續」。



狀態讀取 (State Retrieval) : Orchestrator 從 Session State 中讀取之前存放的「書籤」。

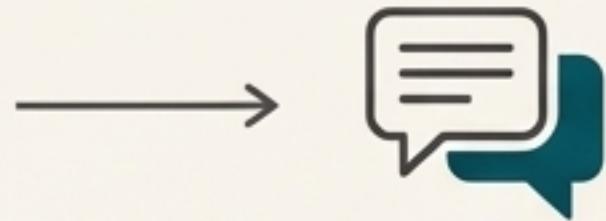
```
context = state.get("jira_pagination_cursor")  
→ 取得 "page_2_token_xyz"
```

建立新任務 (New Task) : Orchestrator 再次呼叫 Jira Agent，但這次傳入了新的參數。

```
search_jira(project="PTE", ...,  
cursor="page_2_token_xyz")
```



Jira Agent (Stateless)
它不需要記得之前的任何事。
它只根據收到的 `cursor` 參數，
向 Jira API 請求第二頁的資料。



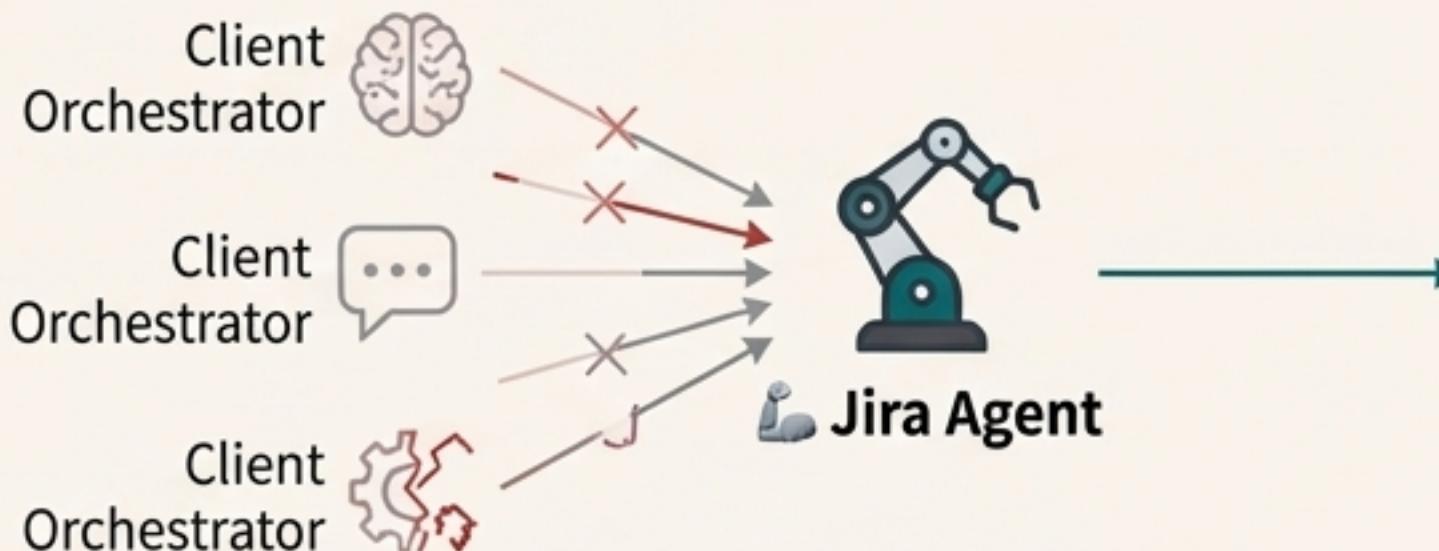
最終回應：Agent 整理並回傳剩下的 40 筆 issues。

結論

狀態 (State) 由 Orchestrator 負責管理，而專家 Agent (Expertise Agent) 保持無狀態 (Stateless)。這種解耦設計讓專家 Agent 可以被輕易地重用與擴展。

服務治理：如何確保我們的 Agent 被正確使用？

核心挑戰 (The Governance Challenge)



我們的算法：「對方詮算密僕」

- 問題: 當我們的 Jira Agent 被公司內其他部門的 Orchestrator 調用，我們無法保證他們的實作能正確處理我們回傳的 `metadata` 與 `cursor`。
- 風險: 對方實作不佳，導致使用者體驗差，最終卻怪罪我們的 Jira Agent 不可靠。

我們的解決方案: "指導性產物" (Instructional Artifact) 設計模式

- 原理: 與其期待 Client Agent 自己弄懂邏輯，不如在 Artifact 中直接用自然語言「教」它下一步該怎麼做。
- 效果: 即使對方的 Orchestrator 邏輯很簡單，只要它的 LLM 能讀懂這段指示，就能大幅提高正確處理分頁的機率。這是一種「防呆」設計。

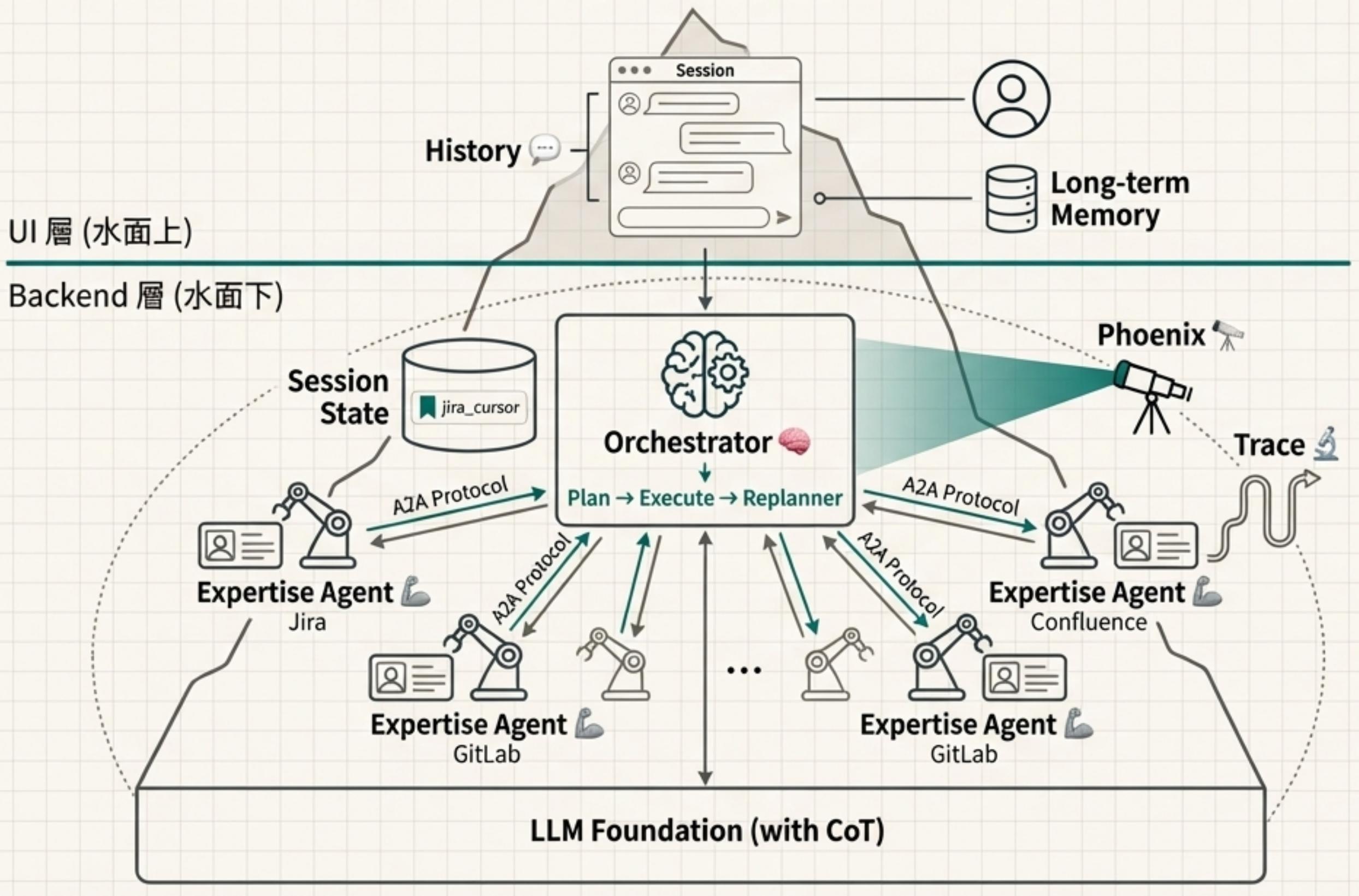
```
{  
  "data": [ ... 10 issues ... ],  
  "metadata": {  
    "has_more": true,  
    "next_cursor": "page_2_token_xyz"  
  },  
  // 關鍵！給對方 LLM 看的「操作說明書」  
  "orchestrator_instruction": "I found 50 issues but only  
  returned 10. Ask the user: 'There are 40 more issues, would  
  you like to see them?'. If yes, call the 'search_jira' tool again  
  with parameter 'cursor' set to 'page_2_token_xyz'."  
}
```

NEW



LLM

總結：我們的 Multi-Agent 系統完整藍圖



圖說

這張圖定義了我們的共同語言與標準作業程序。從使用者的一個請求開始，流經 **Orchestrator** 的規劃，委派給無狀態的專家，並在 **Phoenix** 的全程監控下完成任務，同時透過 **State Handoff** 管理複雜的跨輪對話。

行動項目與下一步 (Next Steps)

目標：將今日達成的共識，轉化為具體的開發實踐。

1

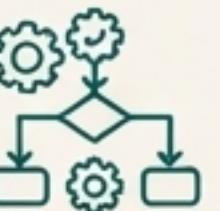


語言與規範定案 (Codify the Language)

任務：將本簡報中的術語定義（特別是 Slide 4, 5, 12）正式寫入團隊的開發 Wiki。

負責人：[Tech Lead Name]
完成時間：本週末前

2



技術框架決策 (Framework Decision)

任務：技術團隊基於我們的架構需求（Plan-and-Execute, State Management, Observability），在一週內完成對 **Google ADK** 與 **LangGraph** 的最終評估，並做出選擇。

負責人：[Architect Name]
完成時間：下週三

3



可觀測性實作 (Implement Observability)

任務：開發團隊開始研究如何將 **Arize Phoenix**（透過 OpenTelemetry）整合進我們選定的框架中。完成一個 PoC，展示一個完整的 Jira 查詢 Trace。

負責人：[Dev Team Lead]
完成時間：兩週內

參考文獻 (References)

為了確保團隊對技術名詞有共同的理解來源，本簡報引用以下文獻與官方文件：

學術論文與核心概念 (Fundamental Papers)

- [1] Wei, J., et al. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." *Google Research*.
- [2] Yao, S., et al. (2023). "ReAct: Synergizing Reasoning and Acting in Language Models." *Princeton & Google*.

Google 官方規範 (Google Specifications)

- [3] Google Cloud. "Agent-to-Agent (A2A) Communication Protocol Specification."
- [4] Google Cloud. "Agent Development Kit (ADK) Documentation." *GitHub & Google Cloud Architecture Center*.

LangChain/LangGraph 生態系 (LangGraph Ecosystem)

- [5] LangChain AI. "LangChain Concepts: Memory vs. Context." *Official Documentation*.
- [6] LangChain AI. "LangGraph Documentation: Graphs, Nodes, and Edges."
- [7] LangChain AI. "LangGraph Persistence: Managing Thread State."

可觀測性與方法論 (Observability & Methodologies)

- [8] Arize AI. "Phoenix Documentation: LLM Tracing & Evaluation."
- [9] LangChain AI. "Plan-and-Execute Agents: Planning for Complex Tasks."
- [10] OpenTelemetry. "OpenTelemetry Specification for LLM Applications."