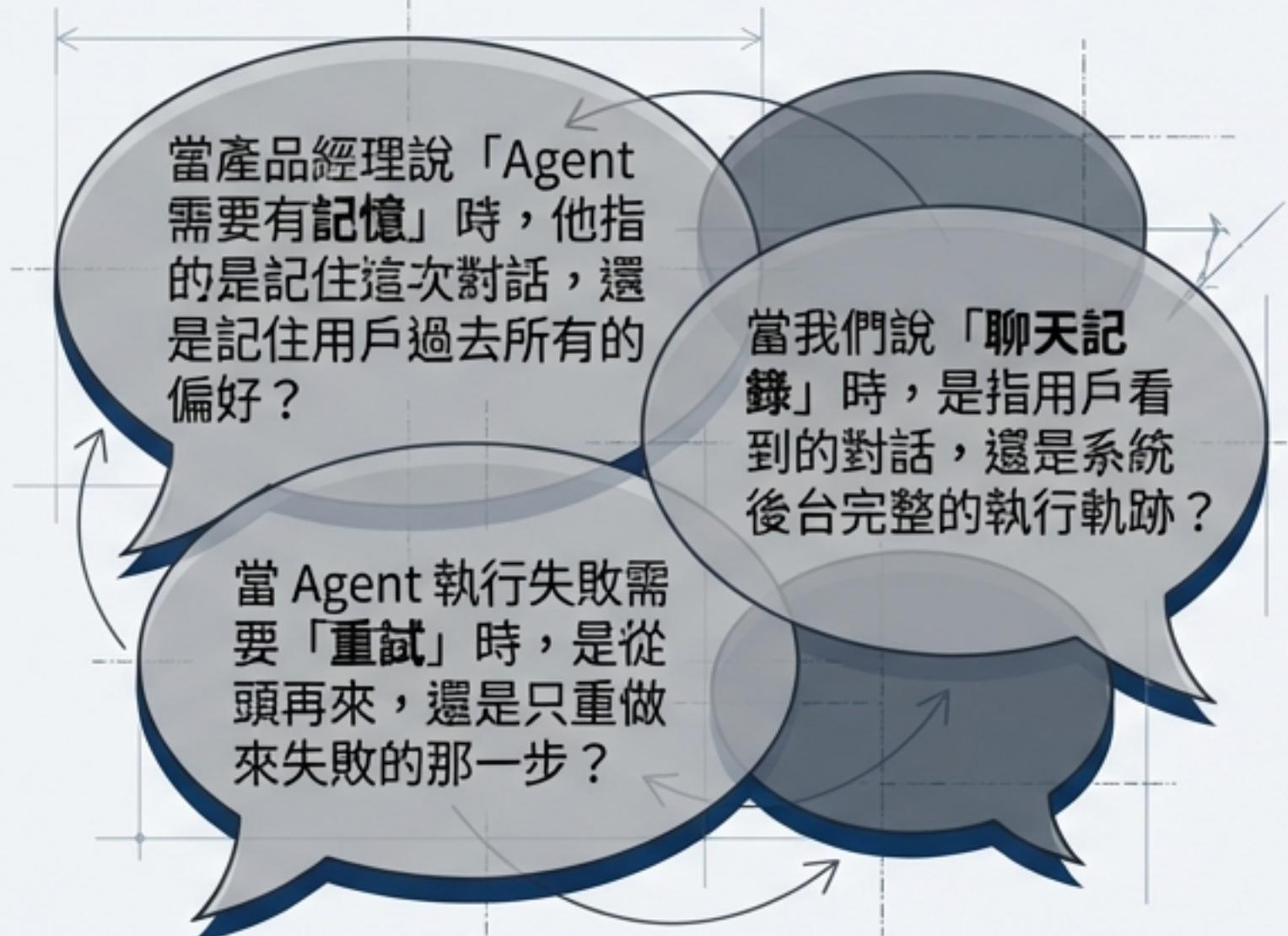


建立共同語言：為何這是我們首要的技術任務

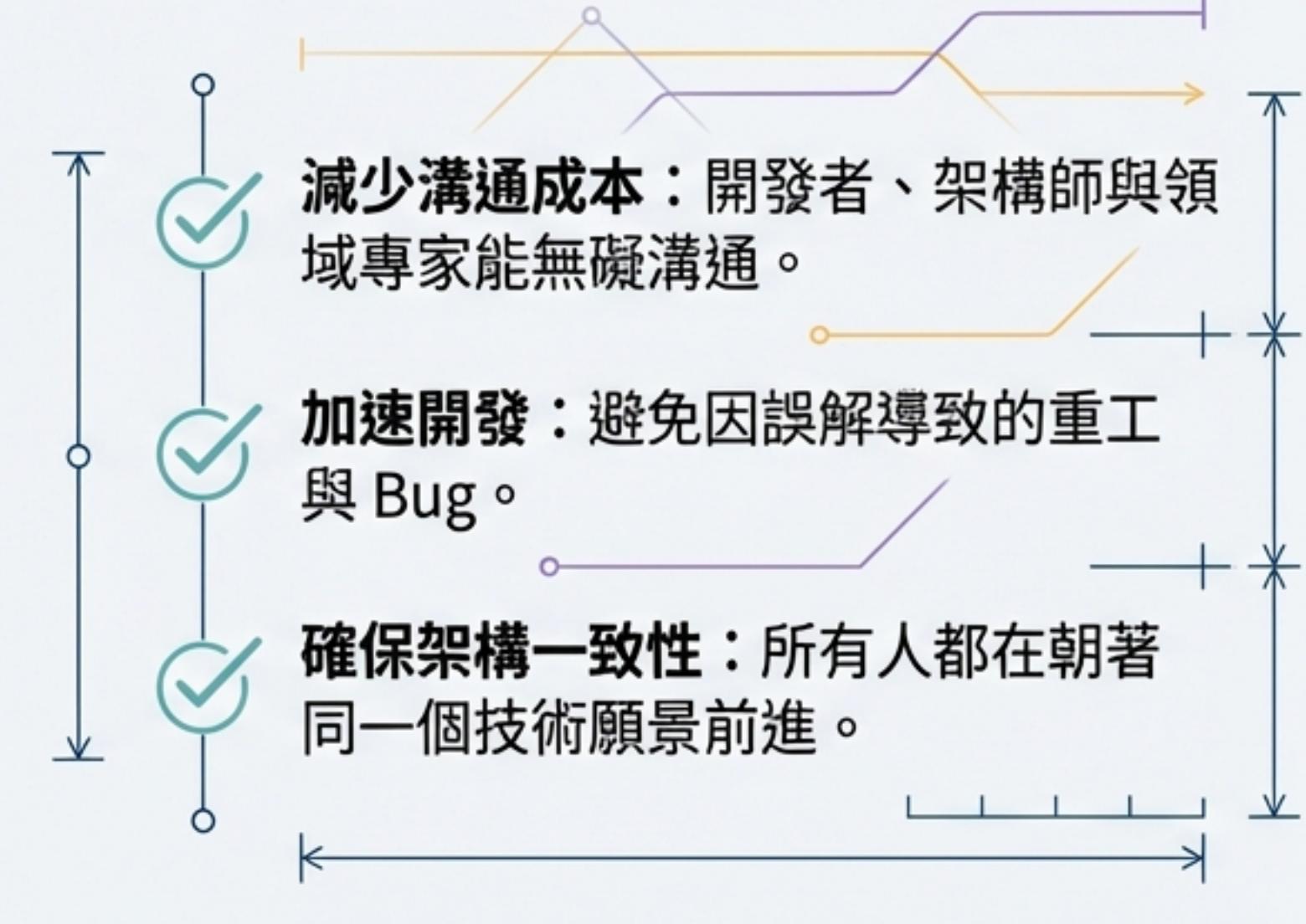
在建構複雜的多代理人系統時，模糊的術語是導致開發摩擦、需求誤解與架構風險的根源。

我們的目標是確立一套精確、無歧義的團隊共同語言 (Ubiquitous Language)。

目前的混亂狀態 (The Chaos)



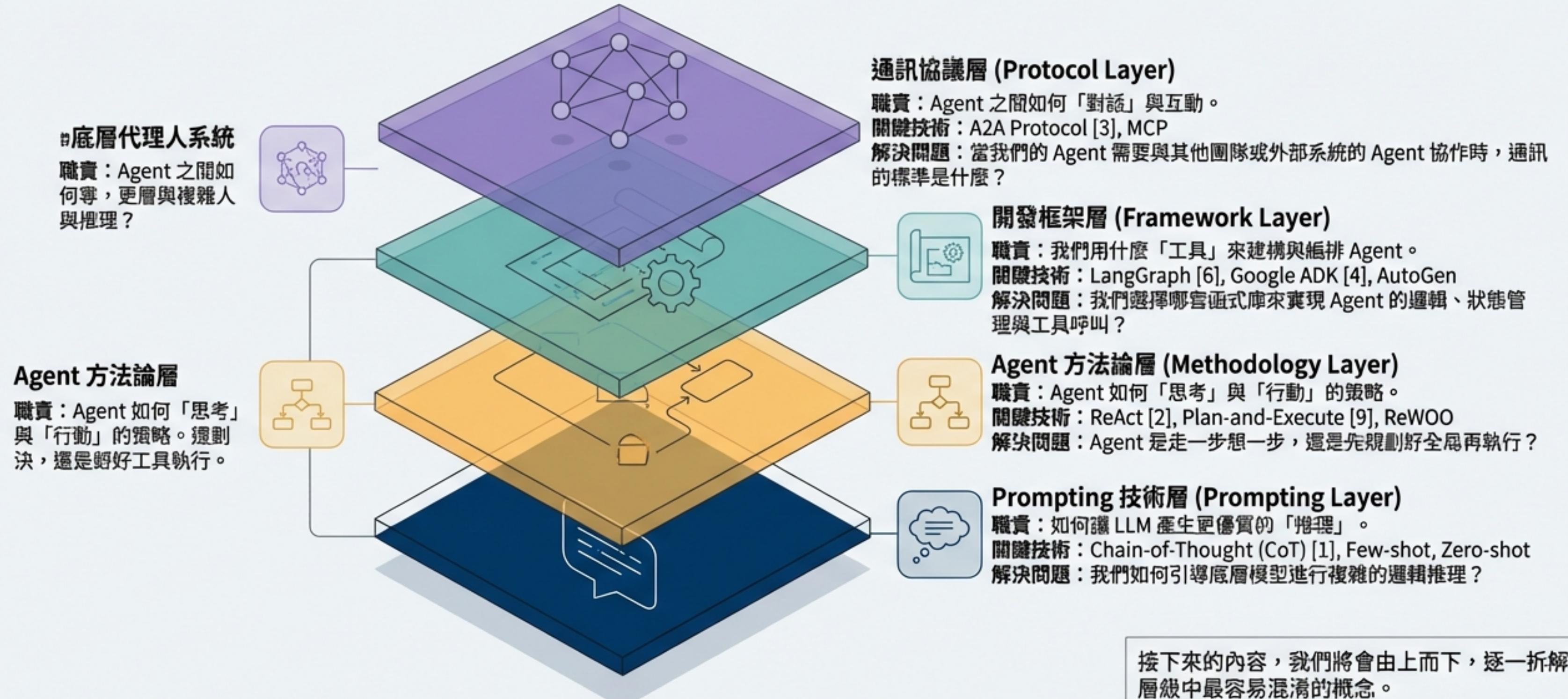
我們的目標：精確的共同語言 (The Clarity)



本簡報將定義一套標準術語，作為我們團隊協作的基礎。

我們的地圖：多代理人系統的四層技術棧

要理解代理人系統，我們必須分層思考。從底層的 LLM 推理，到頂層的跨代理人通訊，每一層都有其特定的概念與工具。



核心釋疑 (1)：記憶的「冰山模型」

同仁常問：「Agent 執行任務的過程（思考、調用工具）算不算記憶？」

答案：算！但它屬於水面下的「短期工作記憶」。

對話記憶 (Chat History / Context) ——

餐廳外場服務生與客人的對話紀錄。

幫我查天氣並做成報告。

這是您的報告檔案。

維持對話連貫性，給 LLM
理解上下文。乾淨、簡潔。

‘Session History’,
‘Messages’ [5]

執行記憶 (Execution Trace / State) ——

廚房裡的備料單、操作步驟與廚師的
內心戲。

‘Thought’：“用戶要天氣報告，我需要先查 API...” (使用 CoT [1])

→ ‘Action’：`call_weather_api(location="Taipei")`

→ ‘Observation’：{ "temp": 25, "humidity": 80 }

→ ‘Thought’：“數據拿到了，現在呼叫報告生成工具...”

→ ‘Action’：`generate_report(...)`

讓 Agent 能自我修正、除錯。
繁瑣但至關重要。

‘Trace’，‘State’，
‘Scratchpad’

將兩者分開是為了避免用繁瑣的執行細節「污染」LLM 的上下文視窗 (Context Window)，同時確保 Agent 的容錯與可除錯性。

記憶術語對照表：從使用者到系統

讓我們將「記憶」的三個層級對應到具體的技術術語，確保我們在討論時指的是同一件事。

	層級	通俗說法	Google ADK 術語 [4]	LangGraph 術語 [7]	定義與範圍
	L1: 長期記憶 (Long-term)	用戶畫像 用戶偏好	Memory	Store (Cross-thread)	跨越不同對話，系統永久記得的資訊。（例如：用戶部門、輸出格式偏好）
	L2: 對話記憶 (Contextual)	聊天記錄 (水面上)	Session Events	Thread Messages	用戶開啟的一次完整對話視窗內，所有「公開」的問與答歷史。
	L3: 執行記憶 (Process/State)	執行過程 (水面下)	Session State Task Trace	State (Schema) Trace	單次任務執行期間，Agent 內部的暫存變數與思考軌跡，任務完成即歸檔。

團隊共識

- 用 戶 體 驗，我 們 討 論 L1 Memory 和 L2 History。
- 系 統 除 錯 與 開 發，我 們 討 論 L3 State 和 Trace。

全鏈路可觀測性：為我們的 Agent 裝上 X 光機

核心挑戰

LLM Agent 本質上是不確定性的 (Stochastic)。當它回答錯誤時，我們如何知道是哪個環節出了問題？

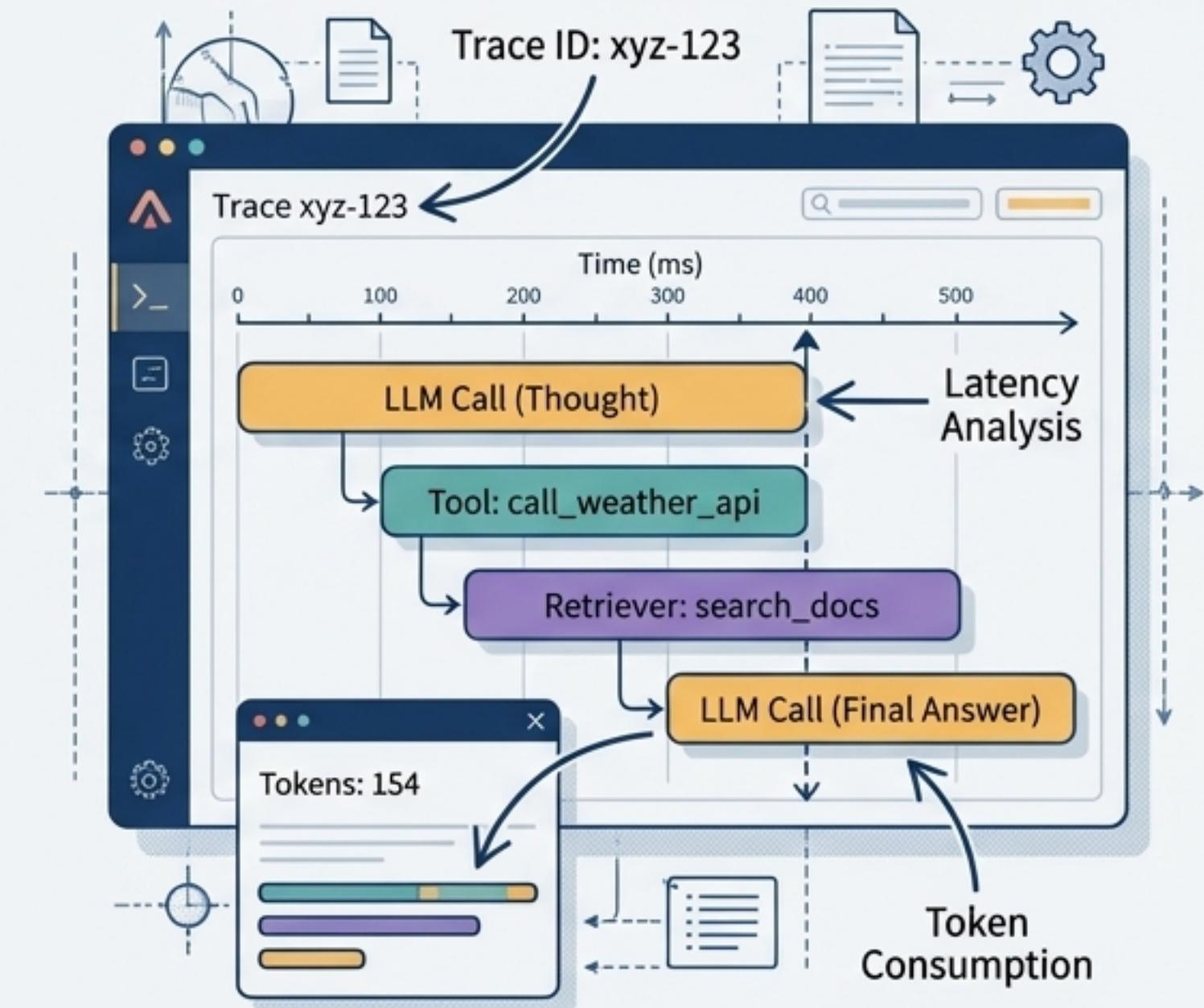
- 是 Prompt 沒寫好？
- 是 RAG 檢索的資料不對？
- 是 Tool 的參數傳錯了？
- 還是 Agent 的「思考」邏輯本身就錯了？

解決方案

導入 Arize Phoenix，將執行軌跡 (Trace) 完全視覺化。

為什麼需要 Phoenix？

- **透視執行過程**：將上一頁提到的「水面下」的‘Trace’從抽象的 Log 轉化為可互動的流程圖 [8]。
- **精準除錯**：當用戶回報問題時，我們能立即調閱該次請求的 Trace ID，快速定位是‘Thought (推理)’、‘Action’ (工具) 還是‘Observation (結果)’環節出錯。
- **效能與成本分析**：清楚看到每一個步驟 (Span) 的耗時 (Latency) 與 Token 消耗，為後續優化提供數據依據。
- **支援開放標準**：基於 OpenTelemetry [10]，能與現有監控系統整合。



Phoenix 將 Agent 的每一步思考與行動都記錄下來，讓我們不再面對一個難以理解的黑盒子。

核心釋疑 (2) : Chain-of-Thought (CoT) vs. Agent Trace

這是最容易混淆的技術點。CoT 只是 Agent 「思考」的技術，而 Trace 是記錄其「完整一生」的日記。

Chain-of-Thought (CoT) - 純粹的「內心獨白」



L1 - Prompting 技術 [1]

革質：純文字推理，不涉及與外部世界的互動。

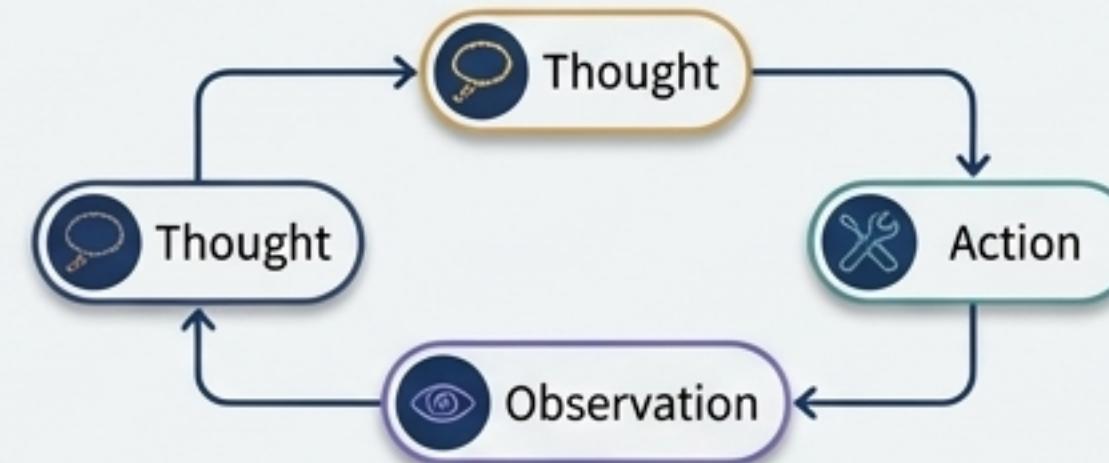
Input: "用戶問台北天氣"

CoT Prompt: "Let's think step by step..."

LLM Output: "首先，我需要確定使用哪個天氣 API。然後，我需要從用戶輸入中提取城市名稱『台北』。最後，我將呼叫 API 並格式化輸出。"

✗ 沒有工具呼叫，✗ 沒有外部回饋。

Agent Trace - 思考 + 行動 + 觀察的「完整軌跡」



L3 - 執行紀錄

革質：Agent 執行任務的完整事件序列。

1. Thought: "我需要查天氣..." (👉 CoT 在這裡被使用)
2. Action: call_weather_api(city="台北")
3. Observation: {"temp": 25}
4. Thought: "數據已取得，可以回覆用戶了。"
5. Final Answer: "台北現在是 25°C。"

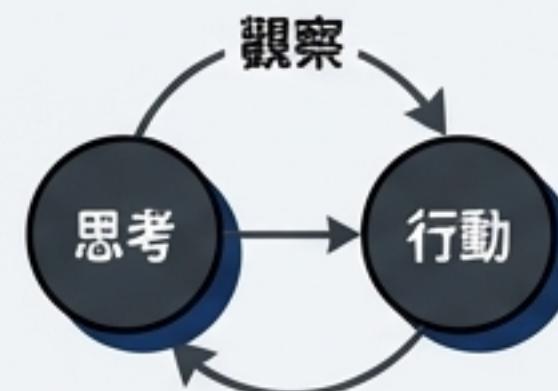
✓ 包含 Thought, Action, Observation 的完整循環。

$$\text{Agent Trace} = \sum(\text{Thought [using CoT]} + \text{Action} + \text{Observation})$$

L2 方法論：Agent 如何規劃它的行動策略

我們不只關心 Agent 的單一步驟，更關心它解決問題的整體策略。這決定了我們系統的穩健性與可控性。

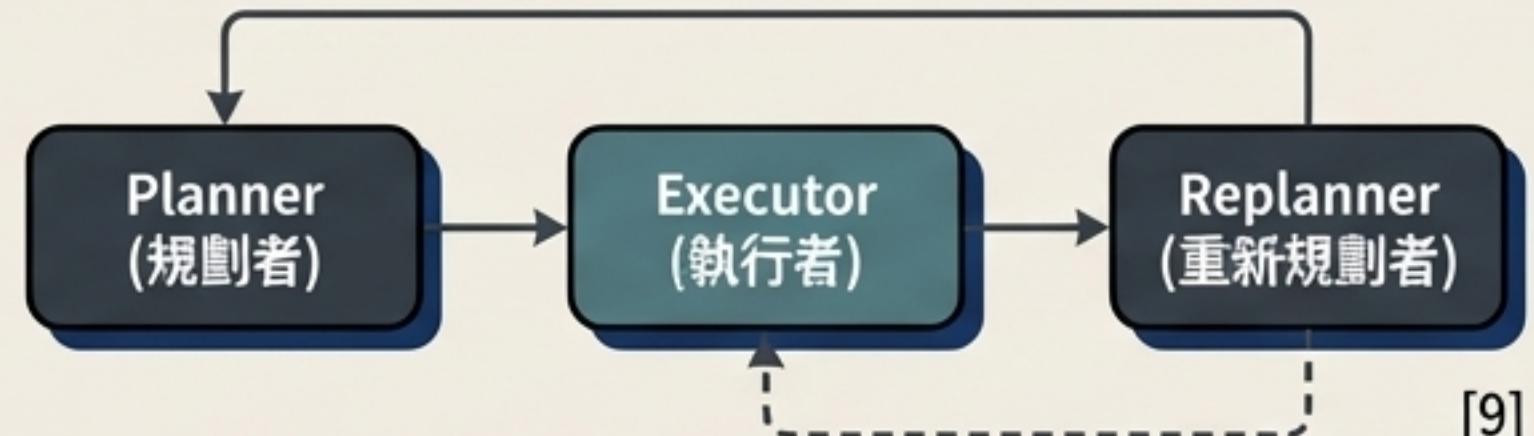
策略 A：ReAct (Reason + Act)



思考 → 行動 → 觀察 → (再)思考... [2]

- 優點：
 - 即時適應性強，對環境變化反應快。
- 缺點：
 - 在長鏈任務中容易「迷失方向」。
 - LLM 呼叫頻繁，成本與延遲較高。
- 適用場景：
 - 簡單、需要即時反饋的任務。

策略 B：Plan-and-Execute (規劃與執行) ★ 專案建議採用



- 優點：
 - 結構清晰，適合複雜、多步驟的任務。
 - 執行階段可使用更小、更快的模型或確定性程式碼，節省成本。
 - 天然地支援「用戶介入修正」的場景。

我們將採用 Plan-and-Execute 作為核心架構，以確保複雜任務的執行成功率與可維護性。

Agent 角色定義：誰是專案經理，誰是專家

在我們的 Plan-and-Execute 架構中，我們需要明確定義不同 Agent 的職責分工。



角色 1 : Orchestrator / Supervisor (專案經理)

- 對應 LangGraph 術語：‘Supervisor’
- 對應 A2A 術語：‘Orchestrator Agent’ / ‘Client Agent’ [3]
- 核心職責：
 - 理解意圖：接收用戶的初始請求。
 - 規劃與拆解：扮演 Planner 的角色，將複雜任務拆解成多個步驟。
 - 委派與協調：根據每個步驟的需求，選擇並呼叫對應的 Expertise Agent。
 - 整合結果：將各個專家的產出匯總成最終答案。

角色 2 : Expertise Agent / Worker (領域專家)

- 對應 LangGraph 術語：‘Worker Agent’ / ‘Node’
- 對應 A2A 術語：‘Remote Agent’ [3]
- 核心職責：
 - 執行專門任務：只專注於一項特定技能 (Skill) 或工具 (Tool)。
 - 提供能力說明：透過 ‘Agent Card’ [3] 或 Tool Definition，告訴 Orchestrator 「我會做什麼」。
- 設計原則：通常設計為無狀態 (Stateless)，接收輸入、完成任務、返回輸出，不保留長期記憶。

L3 開發框架：我們的開發工具箱選擇

方法論與角色定義完畢後，我們需要選擇具體的程式框架來實現它。以下是我們主要的兩個選項。

面向 (Dimension)	Google ADK (Agent Development Kit) 	LangGraph 
設計哲學	與 Google Cloud 生態 (Vertex AI, Gemini) 深度整合，專為企業級對話服務設計。	基於圖 (Graph) 的工作流程設計，將 Agent 互動視為有向圖中的節點與邊 [6]。
核心抽象	Session , Event , State , Memory , Runner [4]	StateGraph , Node , Edge , Thread , Checkpoint [6]
狀態管理	透過 SessionService 和 State 結構化管理，適合對話密集型應用。	極度靈活。透過中央 StateGraph 和 Checkpointer 機制，可實現複雜的狀態轉換與回滾。
學習曲線	中等。需熟悉 Google Cloud 的服務與概念。	較高。需要先理解「圖」的理論與狀態機的概念，才能靈活運用。
最適場景	需要與 Google Cloud 服務緊密整合、對話流程標準化的專案。	需要精細控制複雜工作流程、條件分支、循環與 Agent Handoff 的專案。

技術團隊需要評估哪一個框架的「狀態管理」機制更符合我們 Plan-and-Execute 架構中對 State 和 Trace 的精細化管理需求。

L4 通訊協議：當 Agent 需要跨越系統邊界

A2A 協議不是另一個開發框架，而是 Agent 之間的「通用語言」，讓不同系統、不同團隊開發的 Agent 能夠互相溝通。

什麼是 A2A (Agent-to-Agent) Protocol？

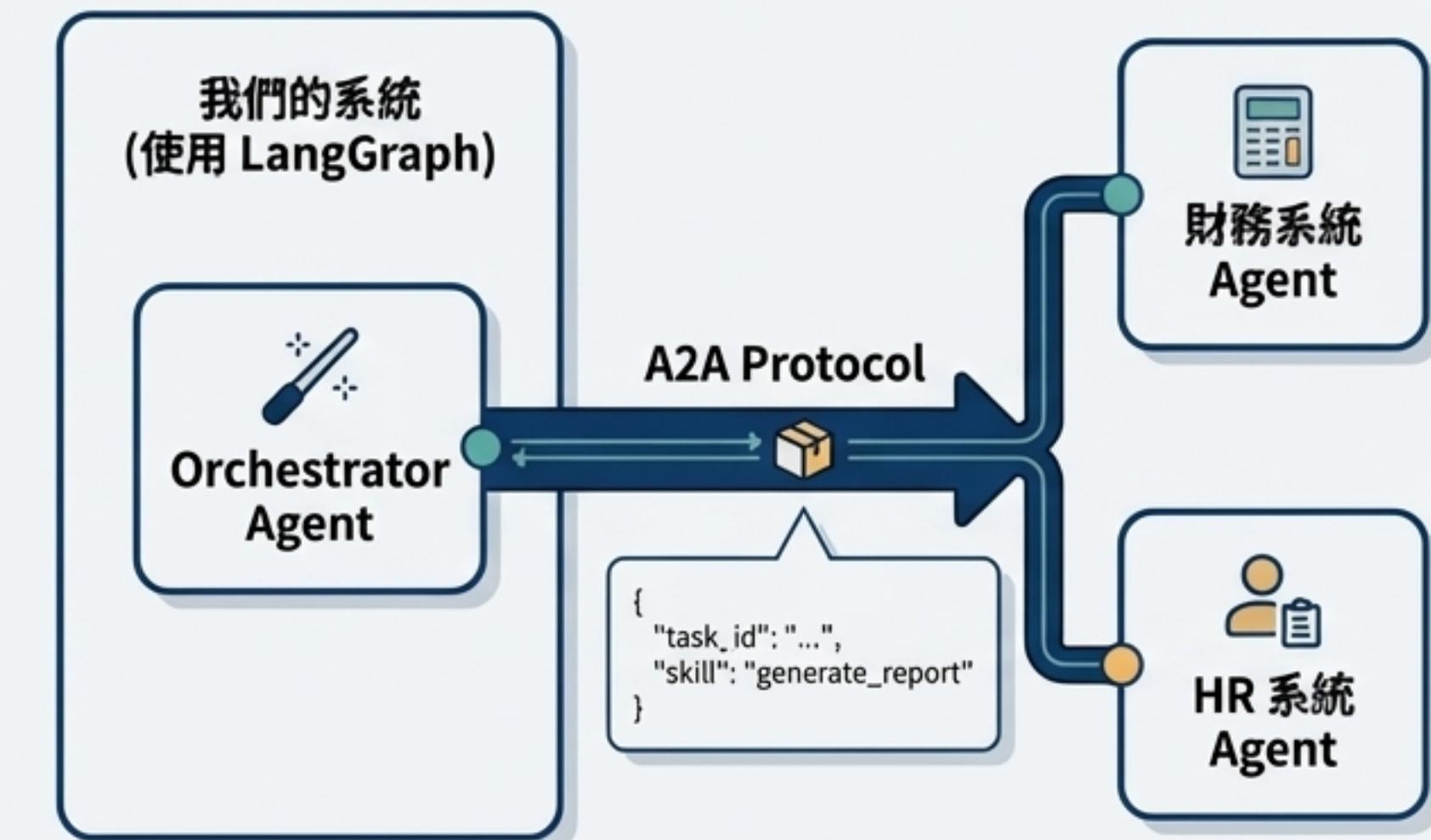
- 它是一個通訊標準，專注於定義 Agent 之間「如何通訊」，而非「如何決策」[3]。
- 它不規定你用 LangGraph 還是 ADK 開發，任何遵守協議的 Agent 都可以加入網絡。

A2A 核心概念

- **Agent Card**：每個 Agent 的「名片」，描述其身份、能力 (`Skills) 與聯絡方式 (`Endpoint) [3]。Orchestrator 透過讀取 Agent Cards 來發現並選擇合適的專家。
- **Task**：一個標準化的工作單元，擁有自己的生命週期狀態（如 `submitted`、`working`、`completed`、`failed`）。
- **Message**：在一個 Task 內部，Client Agent 和 Remote Agent 之間的通訊輪次。

何時需要考慮 A2A？

- 當我們的系統需要與公司內其他獨立的 Agent 服務進行協作時。
- 當我們希望將某些專業 Agent 作為標準服務發布，供其他團隊調用時。



我們的共同語言 v1.0：專案術語定義總表

從今天起，當我們在會議、文件和程式碼中溝通時，請使用以下標準術語。

方法論層 (Methodology)

- “Plan-and-Execute”: 我們採用的核心策略：`Planner` → `Executor` → `Replanner`。
- “ReAct”: 一種備選策略：`Thought` → `Action` → `Observation`。

執行與記憶層 (Execution & Memory)

- “Trace / Trajectory”: Agent 完整的執行軌跡（包含 `Thought`, `Action`, `Observation`），是我們用 Phoenix 觀測與除錯的對象。
- “CoT (Chain-of-Thought)": 僅指 Thought 步驟中使用的純文字推理技術。
- “State”: Agent 在執行單一任務時的短期工作記憶（水面下）。
- “Checkpoint”: LangGraph 中 State 的狀態快照，用於持久化與回滾。

溝通範例

✗ 模糊說法: “把聊天記錄傳給下一個 Agent。”

對話層 (Conversation)

- “Session / Thread”: 用戶開啟的一個對話視窗/線程（水面上）。
- “Messages / History”: Session 內的對話歷史。
- “Memory / Store”: 跨 Session 的長期記憶。

Agent 與任務層 (Agent & Task)

- “Supervisor / Orchestrator”: 負責規劃和分派的主控 Agent。
- “Worker / Remote Agent”: 執行特定工作的專業 Agent。
- “Tool / Skill”: Agent 的特定功能。
- “Task (A2A)": 跨 Agent 通訊的標準工作單位。

✓ 精確說法: “將部分的 ‘Thread Messages’ 和 ‘State’ 透過 ‘Handoff’ 傳遞給 ‘Worker Agent’。”

下一步行動 (Action Items)

我們已經建立了共同的認知地圖與語言，現在需要基於此做出關鍵的技術決策。



1. 確認術語表 (For All)

任務：正式將 Slide 11 的「共同語言 v1.0」納入專案的開發規範文件中。

負責人：[Project Manager Name]
完成時間：[Date]



2. 框架決策 (For Tech Leads & Architects)

任務：基於我們對「狀態管理」與「可觀測性」的需求，完成對 Google ADK 與 LangGraph 的最終技術評估 (PoC)。

- 關鍵評估點：哪個框架能更輕易地與 Arize Phoenix 整合，並實現對 `Trace` 和 `State` 的精細化管理？

負責人：[Tech Lead Name]
完成時間：[Date]



3. 觀測性整合 (For Dev Team)

任務：開始研究 Arize Phoenix 的 OpenTelemetry SDK，並準備一個小範例，演示如何追蹤一次完整的 Plan-and-Execute 流程。

負責人：[Senior Developer Name]
完成時間：[Date]

確保我們在寫下第一行 Agent 程式碼之前，整個團隊在架構、術語和工具上完全對齊。