

# 為何我們需要這份共識文件？

當團隊成員討論 Agent 時，  
我們說的是同一件事嗎？

「聊天記錄」：是指「這次對話的  
內容」還是「使用者過去所有的  
偏好」？[5][7]

「Agent 在思考」：是指它在「規  
劃多個步驟 (Planning)」還是單純的  
「文字推理 (Reasoning)」？[2][9]

「任務失敗」：是指 A2A Task  
失敗？還是 LangGraph 的某個  
Node 執行錯誤？

## 預期產出



**確立共同語言 (Ubiquitous Language)**：建立一套跨職能的精確術語，作為開發與驗收的基準。



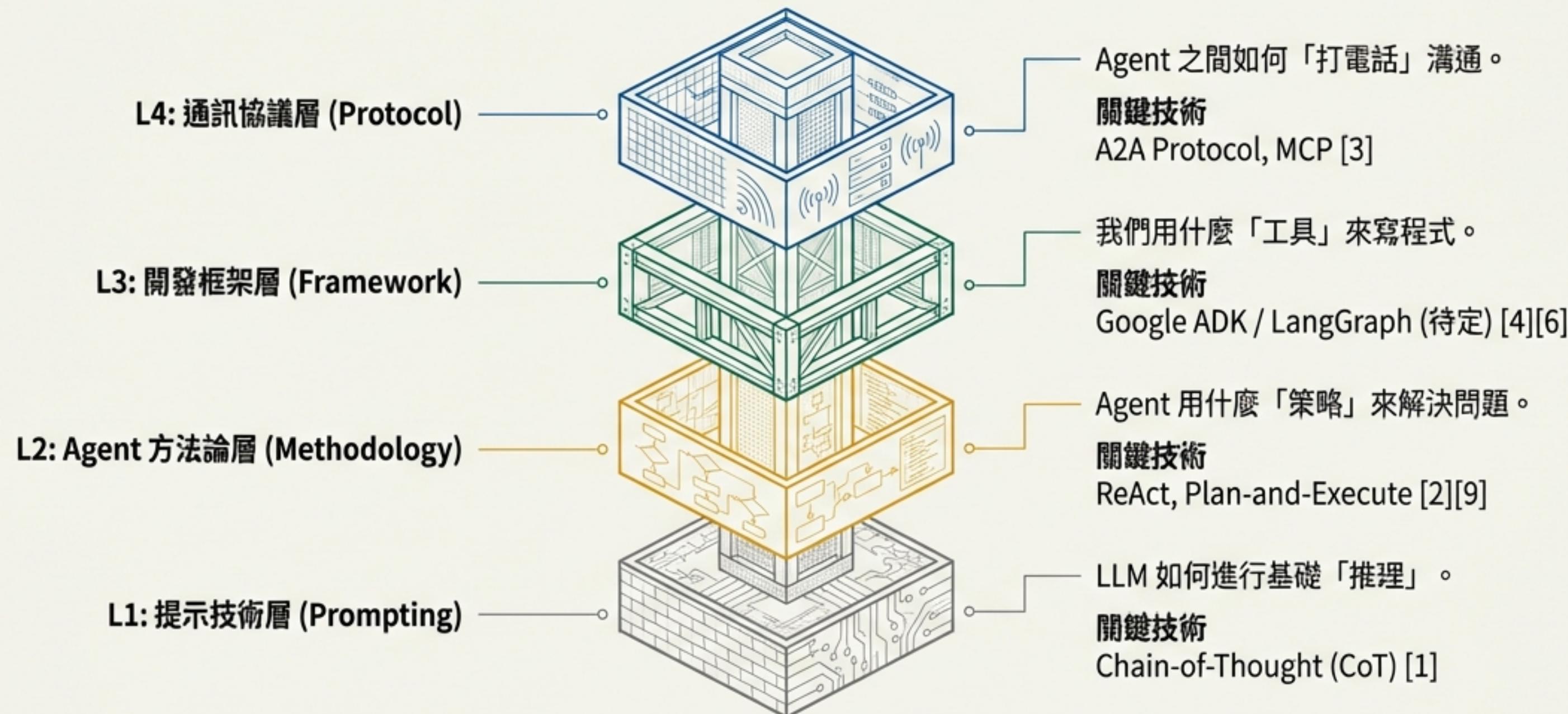
**釐清技術架構**：對齊我們正在建構的系統分層。



**奠定決策基礎**：為接下來的框架選擇 (ADK vs. LangGraph) 提供評估基礎。

# 我們的技術全景圖：Agent 系統的四層架構

Agent 系統並非單一技術，而是由下至上堆疊而成的。後續所有術語都將對應到這個分層。



註解: CoT 是由 Google Research 提出的基礎推理技術 [1]，而 ReAct 則進一步結合了推理與行動 [2]。

# 宏觀視角：使用者互動中的對話與記憶

從「使用者」的角度看，系統記住了什麼？

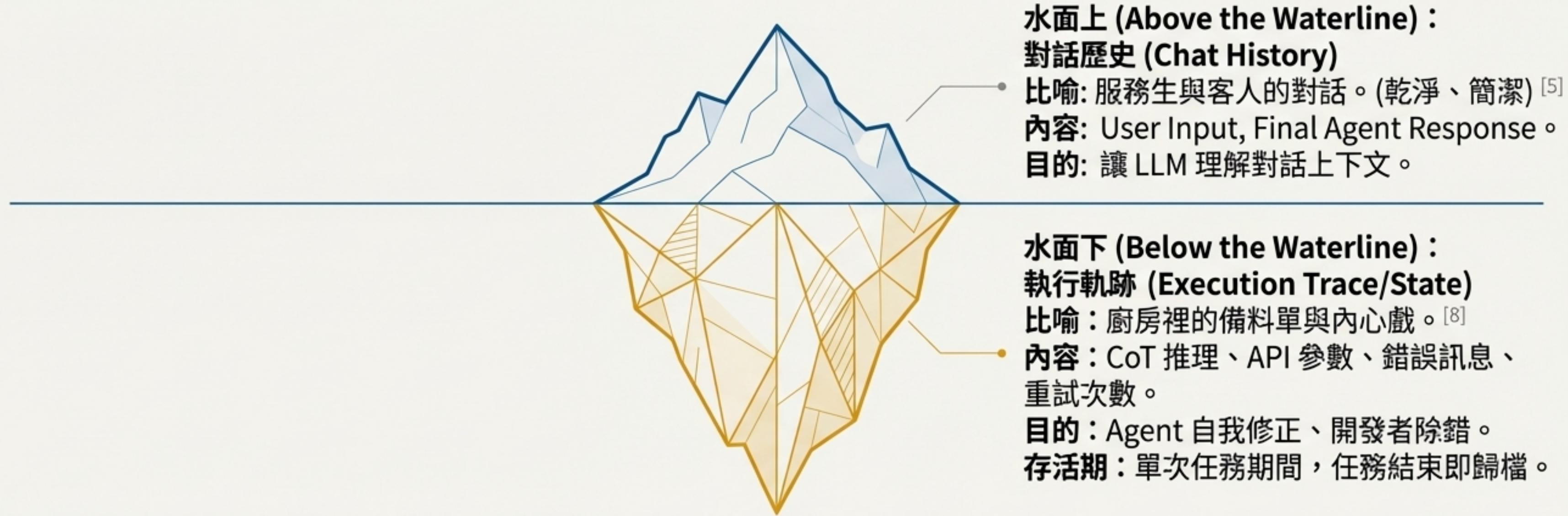
概念	Google ADK 術語 [4]	LangGraph 術語 [7]	定義與範圍
● 會話 (Session)	Session	Thread	用戶開啟的一次完整對話視窗 (由唯一的 thread_id 識別)。
● 歷史 (History)	Events	Messages	這次視窗內「公開」的對話紀錄 (User Input + Agent Final Response)。
● 記憶 (Memory)	Memory	Store (Cross-thread)	跨越不同 Session 的長期用戶畫像 (例如：用戶偏好、所屬部門)。

以後請明確區分「Session History」（這次聊的）和「Long-term Memory」（一直以來記得的）。

# 核心釋疑：記憶的「冰山模型」

關鍵問題：「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「短期工作記憶」，通常不直接呈現給使用者。



**為什麼要分開？**：如果把廚房裡的所有雜事都告訴客人（塞進 Chat History），不僅會讓使用者困惑，也會嚴重污染 LLM 的上下文視窗（Context Window），導致效能下降與成本增加。

# 微觀視角：任務執行與系統軌跡

深入「水面下」，定義開發者需要關注的執行單元。

概念	Google A2A 術語 [3]	LangGraph 術語 [6]	定義
● 任務 (Task)	Task	Run / Job	從接收指令到產出結果的完整週期，具有生命週期狀態。
● 軌跡 (Trace)	Trace / Log	Trace	完整的 Thought → Action → Observation 序列。
● 狀態 (State)	Session State	State (Schema)	執行過程中的暫存變數（例如：past_steps, current_tool）。

開發團隊請注意，當我們說要 Check Log 時，我們看的是 **Trace**；  
當我們說要優化 User Context 時，我們看的是 **History**。

# 全鏈路可觀測性：點亮冰山之下的世界

**核心挑戰：**LLM 具有隨機性 (Stochastic)。當 Agent 回答錯誤時，我們如何知道是 Prompt 寫不好、RAG 沒找到資料，還是 Tool 參數傳錯了？<sup>[8]</sup>

**解決方案：**導入 Arize Phoenix，將 Trace 視覺化。



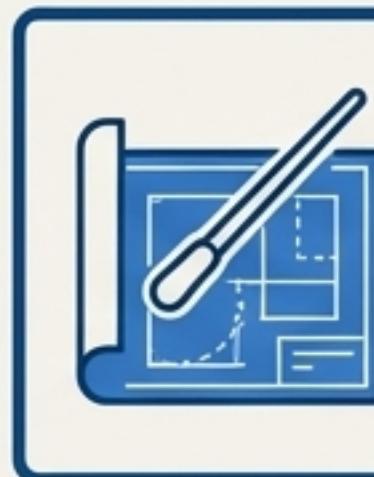
## 為什麼需要 Phoenix？

- **X光般的透視能力：**將 Slide 5 提到的「Trace」轉化為可互動的甘特圖或 DAG 圖，清楚看到每個 Span (LLM 呼叫、Tool 執行) 的輸入、輸出與耗時。<sup>[8]</sup>
- **精準除錯：**當用戶回報問題時，我們能直接調出該 Trace ID，檢查是哪一步「想歪了」 (CoT Error) 還是「做錯了」 (Tool Error)。
- **開放標準：**支援 OpenTelemetry<sup>[10]</sup>，能與現有 DevOps 生態整合。

傳統程式用 Breakpoint 除錯，Agentic Service 則需要 Phoenix 錄下所有的「思考與行動軌跡」。這對上線後的維護至關重要。

# Agent 角色定義：建構我們的虛 nitro team

參考多代理人系統 (Multi-Agent Systems) 的標準職責劃分，進行分工合作。



## Orchestrator / Supervisor (專案經理)

- 職責: 理解用戶意圖、拆解任務、派發工作給專業 Agent、整合最終結果。[3]
- 對應: A2A Orchestrator Agent / LangGraph Supervisor



## Agent Card (履歷表)

職責: 一份機器可讀的能力描述檔 (Metadata)，讓 Orchestrator 知道每個專家「會做什麼」以及「如何呼叫」。[3]



## Expertise Agent / Worker (領域專家)

- 職責: 專注執行特定技能 (Skill/Tool)，例如查詢 Jira。通常設計為無狀態 (Stateless)。
- 對應: A2A Remote Agent / LangGraph Worker Node



## Expertise Agent / Worker (領域專家)

- 職責: 專注執行特定技能 (Skill/Tool)，例如分析財報。通常設計為無狀態 (Stateless)。
- 對應: A2A Remote Agent / LangGraph Worker Node



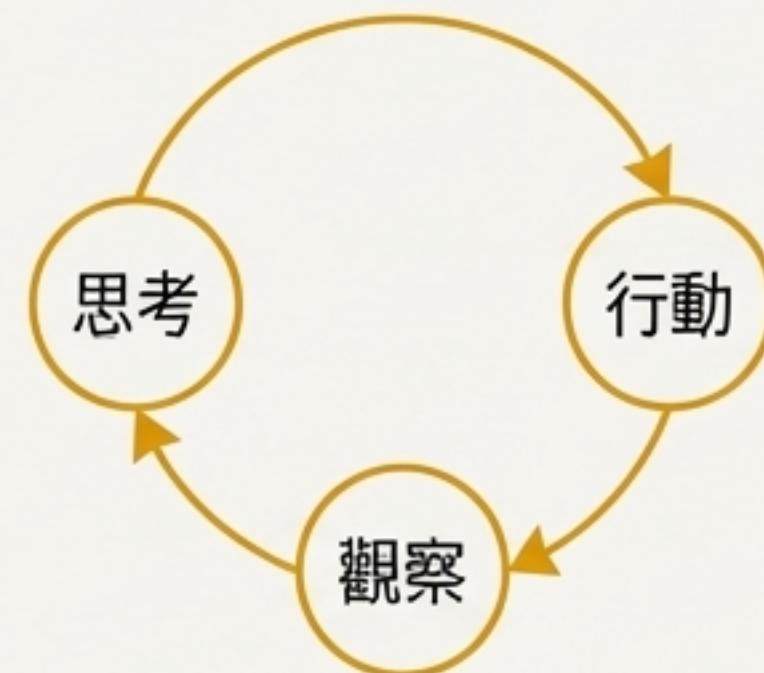
## Expertise Agent / Worker (領域專家)

- 職責: 專注執行特定技能 (Skill/Tool)，例如執行指令。通常設計為無狀態 (Stateless)。
- 對應: A2A Remote Agent / LangGraph Worker Node

# Agent 方法論：我們如何策略性地思考與行動

選擇最適合複雜、可修正任務的執行策略。

## ReAct (Reason + Act)



## Plan-and-Execute (規劃與執行) ★ 本專案選用

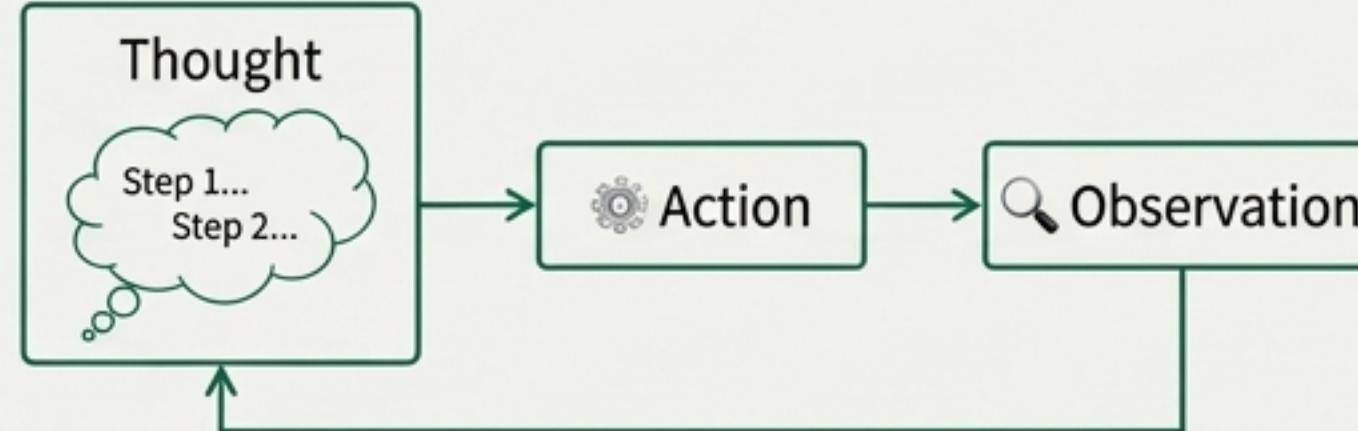


- 優點：即時適應性強。
- 缺點：步驟多時容易迷失方向，LLM 呼叫頻繁。

**優勢**  
結構清晰，適合複雜任務，能優雅地處理錯誤與使用者修正指令，提升長流程任務的成功率。[9]

# 釐清關鍵誤區：Chain-of-Thought (CoT) vs. Agent Trace

CoT 是「內心獨白」，Trace 是「完整日記」。

Chain-of-Thought (CoT) - 推理技術	Agent Trace - 執行記錄
 <p>Step 1... Step 2...</p>	 <pre>graph LR; Thought[Thought&lt;br/&gt;Step 1...&lt;br/&gt;Step 2...] --&gt; Action[Action]; Action --&gt; Observation[Observation]</pre>
<ul style="list-style-type: none"><li>層級：L1 Prompting 技術。</li><li>定義：LLM 產生答案前的「中間推理步驟」，是純文字的內心思考過程。[1]</li><li>範例：「用戶想查上禮拜，今天是週二，所以日期區間是...」</li><li>不包含：工具調用 (Action) 或外部回饋 (Observation)。</li></ul>	<ul style="list-style-type: none"><li>層級：L3 框架層的產物。</li><li>定義：Agent 執行任務的完整軌跡記錄。</li><li>包含：`Thought` (可能使用 CoT) + `Action` (工具調用) + `Observation` (工具返回結果) 的完整序列。</li></ul>

✗ 錯誤：「這個 Agent 的 Chain-of-Thought 記錄了所有工具調用。」

✓ 正確：「這個 Agent 的 Trace 記錄了所有思考和工具調用。」

✓ 正確：「在 Thought 步驟中，我們使用 CoT prompting 來提升推理品質。」

# 實戰演練 (1/3)：Jira 查詢的規劃階段

情境：

使用者：「請問 Jira 上 PTE 專案上禮拜完成的 issues 有哪些？」

當前時間：2025年 12月 2日（週二）

## Orchestrator (Planner) 的 Trace - CoT 推理過程

1 Thought :

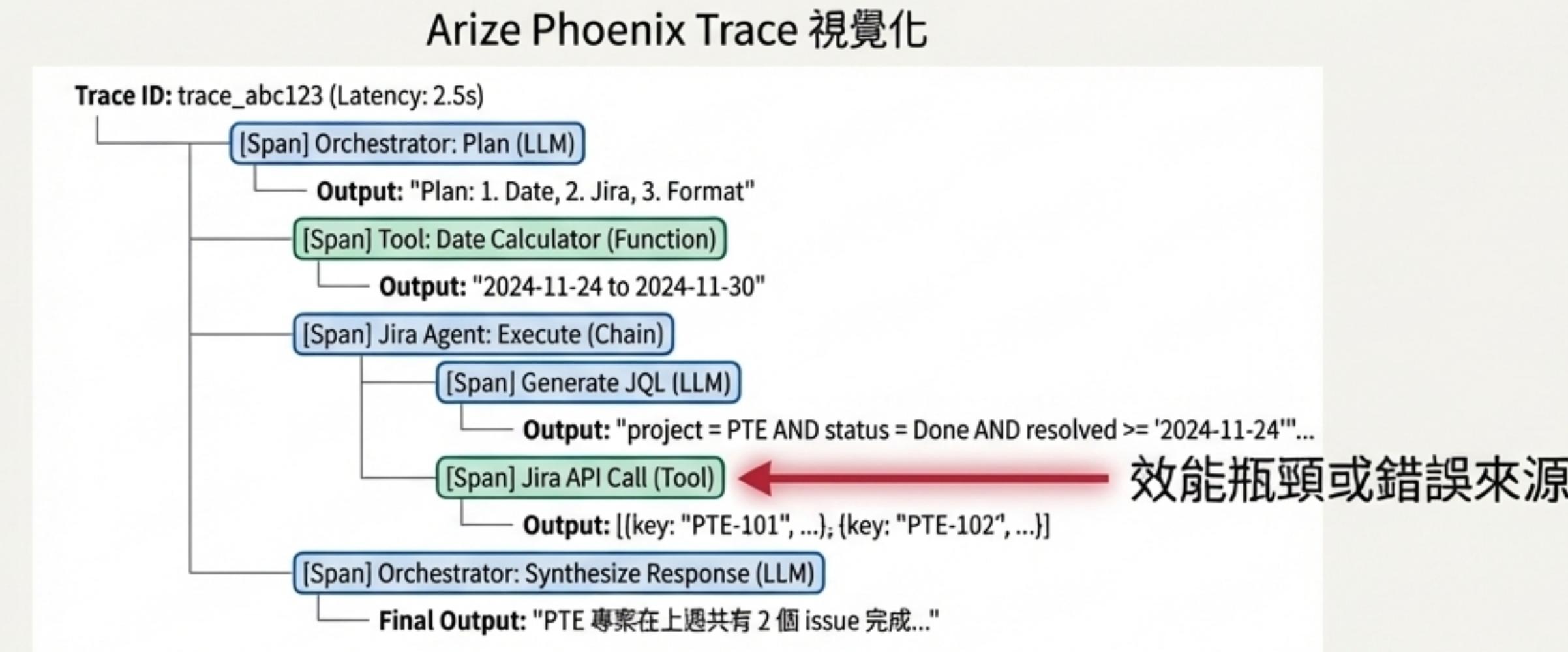
1. 使用者意圖是查詢 Jira issues。
2. 解析實體：專案代碼是 'PTE'。
3. 解析時間：「上禮拜」。今天是 12/2，所以上禮拜的日期範圍是 11/24 到 11/30。
4. 解析狀態：「完成的 (completed)」，對應 Jira 狀態 'Done'。
5. 生成計畫：
  - a. 步驟一：呼叫 Date Calculator 工具確認日期。
  - b. 步驟二：呼叫 Jira Search 工具，傳入解析後的參數。
  - c. 步驟三：將 Jira API 的結果轉換為自然語言。

## State 更新

```
{ user_intent : "query_jira",
  plan : [
    "step_1_date",
    "step_2_jira",
    "step_3_format"
  ],
  entities : {
    "project" : "PTE",
    "status" : "Done",
    "time_expression" : "上禮拜"
  }
}
```

# 實戰演練 (2/3)：執行與可觀測性

Orchestrator (Executor) 依序執行計畫，並委派任務給 Jira Expertise Agent。



如果用戶說「結果不對」，我們可以透過 Phoenix 直接定位到是 `Generate JQL` 這一步的邏輯錯誤，還是 `Jira API Call` 這一步的網路或權限問題。

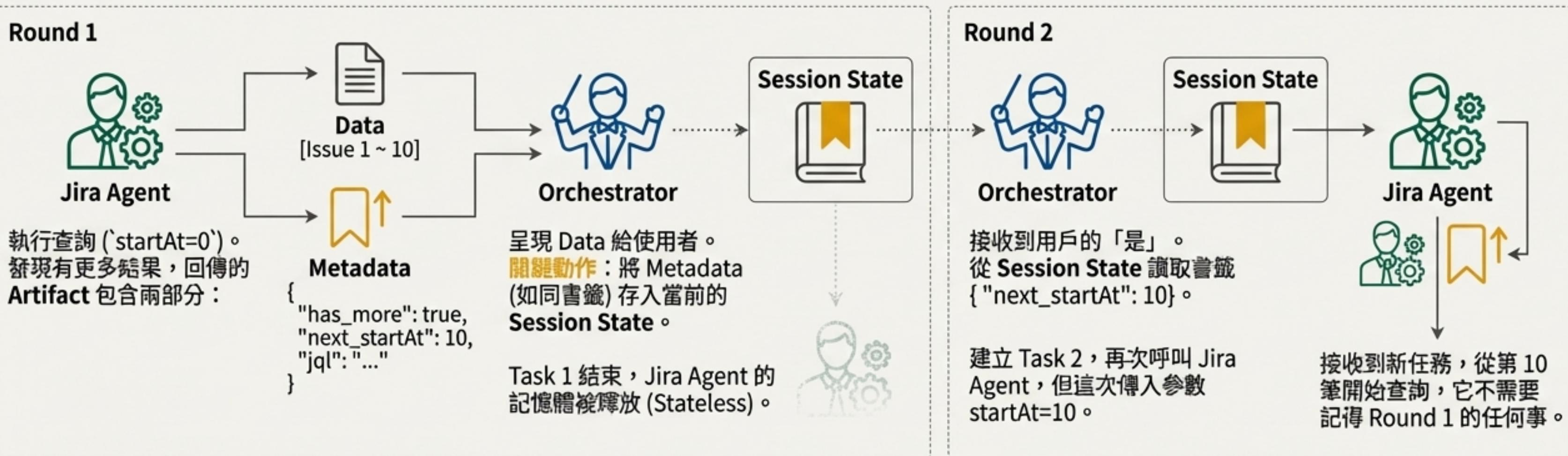
# 實戰演練 (3/3)：處理分頁的狀態交接

## 情境

Agent (Round 1)：「PTE 專案上週有超過 10 個 issue 完成，部分清單如下...。是否需要我列出完整清單？」

使用者：「是。」

## 背後運作機制：狀態交接 (State Handoff)



保持 Expertise Agent 的無狀態，將連續性管理的責任交由 Orchestrator 的 Session State 處理，是兼具彈性與擴展性的最佳實踐。

# 開發框架的選擇：我們的工具箱

根據我們的架構藍圖，選擇最適合的開發框架。

面向	 LangGraph	 Google ADK (Agent Development Kit)
設計哲學	以圖 (Graph) 為基礎的工作流，將 Agent 互動視為節點 (Node) 與邊 (Edge) [6].	專為對話式 Agent 設計，強調 Session 與 Event 管理 [4].
核心抽象	Node, Edge, StateGraph, Checkpoint	Session, Event, State, Memory, Runner
狀態管理	透過中央化的 StateGraph 與 Checkpointer 精細控制狀態，適合複雜流程。	內建 SessionService 管理對話狀態與長期記憶。
生態系	LangChain 生態系，社群龐大，工具整合豐富。	與 Google Cloud (Vertex AI, Gemini) 深度整合，適合企業級合規與大規模部署。
學習曲線	較陡峭，需理解圖論與狀態管理機制。	中等，需熟悉 Google Cloud 的服務與規範。

技術團隊將基於本專案對「狀態管理複雜度」與「企業合規」的需求，在一週內完成評估並提出建議。

# 我們的共同語言 (Ubiquitous Language) v1.0

Goal：在未來的會議、文件和程式碼中，使用以下精確術語。

對話層 (Conversation Layer)	執行層 (Execution Layer)	Agent 層 (Agent Layer)
<b>Session / Thread</b> 一次完整的對話視窗。	<b>Task / Run</b> 一個完整的、有生命週期的工作單元。 <b>Trace</b> 包含 `Thought`、`Action`、`Observation` 的完整執行軌跡，用 Phoenix 觀測。	<b>Orchestrator / Supervisor</b> 負責規劃與分派的主控 Agent。
<b>History / Messages</b> 視窗內「使用者可見」的對話記錄。	<b>State</b> 任務執行期間的「短期工作記憶」或「暫存區」。	<b>Expertise / Worker Agent</b> 執行特定工作的專業 Agent (通常 Stateless)。
<b>Memory / Store</b> 跨 Session 的長期記憶。		<b>CoT (Chain-of-Thought)</b> 僅指 LLM 的「純文字推理」過程。

## 溝通範例 (Communication Example)

✗ 模糊說法：「把聊天記錄傳給下一個 Agent。」

✓ 精確說法：「把 **Session State** 包含在 Task Message 中，透過 **Handoff** 傳給 Remote Agent。」

# 參考文獻 (References)

## 學術論文與核心概念

- [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

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

## LangChain/LangGraph 生態系

- [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

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