01–Part_One (Bilingual)

01–Part_One（中英對照）

Chapter 1: Prompt Chaining

第 1 章：提示串接

Prompt Chaining Pattern Overview

提示串接模式概覽

Prompt chaining, sometimes referred to as Pipeline pattern, represents a powerful paradigm for handling intricate tasks when leveraging large language models (LLMs). Rather than expecting an LLM to solve a complex problem in a single, monolithic step, prompt chaining advocates for a divide–and–conquer strategy. The core idea is to break down the original, daunting problem into a sequence of smaller, more manageable sub–problems. Each sub–problem is addressed individually through a specifically designed prompt, and the output generated from one prompt is strategically fed as input into the subsequent prompt in the chain.

提示串接（有時也稱為流水線模式）是使用大型語言模型（LLM）處理複雜任務的一種強大範式。它不期望 LLM 以單一步驟解出龐大問題，而是主張「分而治之」。核心概念是把原本艱鉅的問題拆成一連串更小、更易管理的子問題。每個子問題用特別設計的提示逐一處理，並把前一步的輸出策略性地作為下一步的輸入。

This sequential processing technique inherently introduces modularity and clarity into the interaction with LLMs. By decomposing a complex task, it becomes easier to understand and debug each individual step, making the overall process more robust and interpretable. Each step in the chain can be meticulously crafted and optimized to focus on a specific aspect of the larger problem, leading to more accurate and focused outputs.

這種序列式處理自然帶來模組化與清晰度。透過拆解複雜任務，每個步驟更容易理解與除錯，整體流程也更穩健、可解釋。鏈中的每一步都能被精細設計與優化，以聚焦於大問題

中的某一面向，進而提高準確度與聚焦程度。

The output of one step acting as the input for the next is crucial. This passing of information establishes a dependency chain, hence the name, where the context and results of previous operations guide the subsequent processing. This allows the LLM to build on its previous work, refine its understanding, and progressively move closer to the desired solution.

每一步的輸出成為下一步的輸入是關鍵。這種資訊傳遞形成依賴鏈，因此得名；先前步驟的脈絡與結果會引導後續處理。它讓 LLM 能建立在前一步成果上，逐步修正理解並更接近目標解答。

Furthermore, prompt chaining is not just about breaking down problems; it also enables the integration of external knowledge and tools. At each step, the LLM can be instructed to interact with external systems, APIs, or databases, enriching its knowledge and abilities beyond its internal training data. This capability dramatically expands the potential of LLMs, allowing them to function not just as isolated models but as integral components of broader, more intelligent systems.

此外，提示串接不只是拆解問題，也能整合外部知識與工具。在每一步，LLM 可以被指示與外部系統、API 或資料庫互動，將能力拓展到訓練資料以外。這大幅擴展了 LLM 的潛力，讓它們不只是孤立模型，而是更大、更智慧系統的關鍵元件。

The significance of prompt chaining extends beyond simple problem-solving. It serves as a foundational technique for building sophisticated AI agents. These agents can utilize prompt chains to autonomously plan, reason, and act in dynamic environments. By strategically structuring the sequence of prompts, an agent can engage in tasks requiring multi-step reasoning, planning, and decision-making. Such agent workflows can mimic human thought processes more closely, allowing for more natural and effective interactions with complex domains and systems.

提示串接的意義不只在於解題，它是建構進階 AI 代理人的基礎技術。代理人可利用提示鏈在動態環境中自主規劃、推理與行動。透過策略性地安排提示序列，代理人能處理需要多步推理、規劃與決策的任務。這類流程更接近人類思考方式，使其在複雜領域與系統中互動得更自然、更有效。

Limitations of single prompts: For multifaceted tasks, using a single, complex prompt for an LLM can be inefficient, causing the model to struggle with constraints and instructions, potentially leading to instruction neglect where parts of the prompt

are overlooked, contextual drift where the model loses track of the initial context, error propagation where early errors amplify, prompts which require a longer context window where the model gets insufficient information to respond back and hallucination where the cognitive load increases the chance of incorrect information. For example, a query asking to analyze a market research report, summarize findings, identify trends with data points, and draft an email risks failure as the model might summarize well but fail to extract data or draft an email properly.

單一提示的限制：對多面向任務而言，以單一複雜提示驅動 LLM 往往效率不佳，模型需同時滿足多重限制與指令，容易出現指令忽略（漏掉部分要求）、脈絡漂移（失去原始上下文）、錯誤擴散（早期錯誤被放大）、需要更長上下文而資訊不足，以及認知負荷過高導致幻覺等問題。例如：要求模型分析市場研究報告、摘要重點、找出趨勢與數據點、再起草一封電子郵件，很可能失敗，因為模型可能只做得好摘要，卻沒能正確提取數據或完成郵件。

Enhanced Reliability Through Sequential Decomposition: Prompt chaining addresses these challenges by breaking the complex task into a focused, sequential workflow, which significantly improves reliability and control. Given the example above, a pipeline or chained approach can be described as follows:

透過序列拆解提升可靠性：提示串接透過把複雜任務拆成聚焦的序列流程來解決上述問題，大幅提升可靠性與可控性。以上述例子而言，可用如下流水線或鏈式流程：

1. Initial Prompt (Summarization): “Summarize the key findings of the following market research report: [text].”The model’s sole focus is summarization, increasing the accuracy of this initial step.

2. Second Prompt (Trend Identification): “Using the summary, identify the top three emerging trends and extract the specific data points that support each trend: [output from step 1].”This prompt is now more constrained and builds directly upon a validated output.

3. Third Prompt (Email Composition): “Draft a concise email to the marketing team that outlines the following trends and their supporting data: [output from step 2].”

4. 第一個提示（摘要）：“Summarize the key findings of the following market

research report: [text].”模型只需專注摘要，能提升第一步的準確度。

5. 第二個提示（趨勢識別）：“Using the summary, identify the top three emerging trends and extract the specific data points that support each trend: [output from step 1].”這個提示更受限，並建立在已驗證的輸出上。

6. 第三個提示（撰寫郵件）：“Draft a concise email to the marketing team that outlines the following trends and their supporting data: [output from step 2].”

This decomposition allows for more granular control over the process. Each step is simpler and less ambiguous, which reduces the cognitive load on the model and leads to a more accurate and reliable final output. This modularity is analogous to a computational pipeline where each function performs a specific operation before passing its result to the next. To ensure an accurate response for each specific task, the model can be assigned a distinct role at every stage. For example, in the given scenario, the initial prompt could be designated as “Market Analyst,”the subsequent prompt as “Trade Analyst,”and the third prompt as “Expert Documentation Writer,” and so forth.

這種拆解讓流程得以更細緻地控制。每一步更簡單、歧義更少，能降低模型認知負荷，提升最終輸出的準確與可靠性。此模組化方式類似計算流程管線：每個函式完成特定操作後把結果傳給下一個。為確保每個任務的正確性，可以在每一階段指定不同角色，例如第一步為「市場分析師」、第二步為「產業分析師」、第三步為「專業文件撰寫者」等。

The Role of Structured Output: The reliability of a prompt chain is highly dependent on the integrity of the data passed between steps. If the output of one prompt is ambiguous or poorly formatted, the subsequent prompt may fail due to faulty input. To mitigate this, specifying a structured output format, such as JSON or XML, is crucial.

結構化輸出的角色：提示鏈的可靠性高度依賴步驟間傳遞資料的品質。如果某一步輸出含糊或格式不良，後續提示可能因輸入有誤而失敗。為降低風險，必須指定結構化輸出格式，例如 JSON 或 XML。

For example, the output from the trend identification step could be formatted as a JSON object:

例如，趨勢識別步驟的輸出可以格式化為 JSON：

```
{  "trends": [
        {
            "trend_name": "AI-Powered Personalization",
            "supporting_data": "73% of consumers prefer to do business with brands that
        },
        {
            "trend_name": "Sustainable and Ethical Brands",
            "supporting_data": "Sales of products with ESG-related claims grew 28% over
        }
    ]
}
```

This structured format ensures that the data is machine-readable and can be precisely parsed and inserted into the next prompt without ambiguity. This practice minimizes errors that can arise from interpreting natural language and is a key component in building robust, multi-step LLM-based systems.

這種結構化格式確保資料可被機器讀取，能精準解析並無歧義地輸入到下一個提示中。這可降低自然語言解讀所造成的錯誤，是打造穩健多步驟 LLM 系統的關鍵要素。

Practical Applications & Use Cases

實務應用與使用情境

Prompt chaining is a versatile pattern applicable in a wide range of scenarios when building agentic systems. Its core utility lies in breaking down complex problems into sequential, manageable steps. Here are several practical applications and use cases:

提示串接是建構代理式系統時極具彈性的模式，可應用於多種情境。其核心價值在於把複雜問題拆成連續且可管理的步驟。以下是幾個實務應用與使用情境：

1. Information Processing Workflows

1. 資訊處理流程

Many tasks involve processing raw information through multiple transformations. For instance, summarizing a document, extracting key entities, and then using those entities to query a database or generate a report. A prompt chain could look like:

許多任務需要對原始資訊進行多次轉換，例如：先摘要文件、再抽取關鍵實體，最後用這些實體查詢資料庫或生成報告。提示鏈可以如下：

- Prompt 1: Extract text content from a given URL or document.
- Prompt 2: Summarize the cleaned text.
- Prompt 3: Extract specific entities (e.g., names, dates, locations) from the summary or original text.
- Prompt 4: Use the entities to search an internal knowledge base.
- Prompt 5: Generate a final report incorporating the summary, entities, and search results.
- 提示 1：從指定 URL 或文件抽取文字內容。
- 提示 2：摘要清理後的文本。
- 提示 3：從摘要或原文抽取特定實體（如人名、日期、地點）。
- 提示 4：用實體搜尋內部知識庫。
- 提示 5：生成包含摘要、實體與搜尋結果的最終報告。

This methodology is applied in domains such as automated content analysis, the development of AI-driven research assistants, and complex report generation.

此方法常用於自動化內容分析、AI 研究助理開發與複雜報告生成。

2. Complex Query Answering

2. 複雜問題回答

Answering complex questions that require multiple steps of reasoning or information retrieval is a prime use case. For example, “What were the main causes of the stock market crash in 1929, and how did government policy respond?”

需要多步推理或資訊檢索的複雜問題是典型用例。例如：「1929 年股市崩盤的主要原因是什麼？政府政策如何回應？」

- Prompt 1: Identify the core sub-questions in the user’s query (causes of crash, government response).
- Prompt 2: Research or retrieve information specifically about the causes of the 1929 crash.
- Prompt 3: Research or retrieve information specifically about the government’s policy response to the 1929 stock market crash.
- Prompt 4: Synthesize the information from steps 2 and 3 into a coherent answer to the original query.
- 提示 1：辨識問題中的核心子問題（崩盤原因、政府回應）。
- 提示 2：檢索或研究 1929 年崩盤原因的資訊。
- 提示 3：檢索或研究政府對 1929 年股災的政策回應。
- 提示 4：綜合第 2、3 步資訊，形成原問題的完整答案。

This sequential processing methodology is integral to developing AI systems capable of multi-step inference and information synthesis. Such systems are required when a

query cannot be answered from a single data point but instead necessitates a series of logical steps or the integration of information from diverse sources.

這種序列處理方法是打造具備多步推理與資訊整合能力 AI 系統的核心。當問題無法由單一資料點回答，而需要一連串推理步驟或整合多來源資訊時，就必須採用此方法。

For example, an automated research agent designed to generate a comprehensive report on a specific topic executes a hybrid computational workflow. Initially, the system retrieves numerous relevant articles. The subsequent task of extracting key information from each article can be performed concurrently for each source. This stage is well-suited for parallel processing, where independent sub-tasks are run simultaneously to maximize efficiency.

例如，一個自動化研究代理人需要生成某主題的綜合報告時，通常會執行混合式工作流程。系統先擷取大量相關文章，接著針對每篇文章抽取關鍵資訊。這一階段適合平行處理，讓各個獨立子任務同時執行以提升效率。

However, once the individual extractions are complete, the process becomes inherently sequential. The system must first collate the extracted data, then synthesize it into a coherent draft, and finally review and refine this draft to produce a final report. Each of these later stages is logically dependent on the successful completion of the preceding one. This is where prompt chaining is applied: the collated data serves as the input for the synthesis prompt, and the resulting synthesized text becomes the input for the final review prompt. Therefore, complex operations frequently combine parallel processing for independent data gathering with prompt chaining for the dependent steps of synthesis and refinement.

但當個別抽取完成後，流程就自然變成序列式。系統需先彙整資料，再整合成草稿，最後再審閱與修訂產出最終報告。這些後續步驟都依賴前一步完成，這正是提示串接的用武之地：彙整資料作為整合提示的輸入，整合後的文字再作為最後審閱提示的輸入。因此，複雜流程常結合平行處理的資料蒐集與提示串接的整合/精煉步驟。

3. Data Extraction and Transformation

3. 資料抽取與轉換

The conversion of unstructured text into a structured format is typically achieved through an iterative process, requiring sequential modifications to improve the accu-

racy and completeness of the output.

將非結構化文本轉換成結構化格式通常是迭代式流程，需要透過序列修改逐步提升輸出準確度與完整性。

- Prompt 1: Attempt to extract specific fields (e.g., name, address, amount) from an invoice document.
- Processing: Check if all required fields were extracted and if they meet format requirements.
- Prompt 2 (Conditional): If fields are missing or malformed, craft a new prompt asking the model to specifically find the missing/malformed information, perhaps providing context from the failed attempt.
- Processing: Validate the results again. Repeat if necessary.
- Output: Provide the extracted, validated structured data.
- 提示 1：嘗試從發票文件抽取指定欄位（如姓名、地址、金額）。
- 處理：檢查是否抽取到所有必要欄位，且格式是否符合要求。
- 提示 2（條件式）：若欄位缺失或格式錯誤，重新設計提示，請模型特別尋找缺失或錯誤資訊，必要時提供先前失敗的脈絡。
- 處理：再次驗證結果，需要時重複。
- 輸出：提供已驗證的結構化資料。

This sequential processing methodology is particularly applicable to data extraction and analysis from unstructured sources like forms, invoices, or emails. For example, solving complex Optical Character Recognition (OCR) problems, such as processing a PDF form, is more effectively handled through a decomposed, multi-step approach.

此序列式方法特別適合用於表單、發票或郵件等非結構化來源的資料抽取與分析。例如處理 PDF 表單等複雜 OCR 任務，更適合採用拆解式、多步驟方法。

Initially, a large language model is employed to perform the primary text extraction from the document image. Following this, the model processes the raw output to normalize the data, a step where it might convert numeric text, such as “one thousand and fifty,”into its numerical equivalent, 1050. A significant challenge for LLMs is performing precise mathematical calculations. Therefore, in a subsequent step, the system can delegate any required arithmetic operations to an external calculator tool. The LLM identifies the necessary calculation, feeds the normalized numbers to the tool, and then incorporates the precise result. This chained sequence of text extraction, data normalization, and external tool use achieves a final, accurate result that is often difficult to obtain reliably from a single LLM query.

一開始，LLM 先從文件影像中進行文字抽取。接著模型會對原始輸出進行資料正規化，例如將「one thousand and fifty」轉為數字 1050。LLM 在精準數學計算上往往有限，因此下一步可以把算術交給外部計算工具。LLM 先辨識需要計算的內容，再將正規化數字送往工具，並把精確結果納入輸出。這種文字抽取、資料正規化與外部工具使用的鏈式流程，能得到單次 LLM 查詢難以可靠達成的精確結果。

4. Content Generation Workflows

4. 內容生成流程

The composition of complex content is a procedural task that is typically decomposed into distinct phases, including initial ideation, structural outlining, drafting, and sub–sequent revision

複雜內容的撰寫通常是程序式工作，會拆成多個階段，例如發想、結構大綱、起草與後續修訂。

- Prompt 1: Generate 5 topic ideas based on a user’s general interest.

- Processing: Allow the user to select one idea or automatically choose the best one.

- Prompt 2: Based on the selected topic, generate a detailed outline.
- Prompt 3: Write a draft section based on the first point in the outline.
- Prompt 4: Write a draft section based on the second point in the outline, providing the previous section for context. Continue this for all outline points.
- Prompt 5: Review and refine the complete draft for coherence, tone, and grammar.
- 提示 1：依使用者的興趣產生 5 個主題構想。
- 處理：讓使用者挑選一個，或自動選出最佳。
- 提示 2：基於所選主題產生詳盡大綱。
- 提示 3：根據大綱第一點撰寫段落草稿。
- 提示 4：根據大綱第二點撰寫段落草稿，並提供前一段作為脈絡，依序完成各段。
- 提示 5：審閱與修訂完整草稿，調整連貫性、語氣與文法。

This methodology is employed for a range of natural language generation tasks, including the automated composition of creative narratives, technical documentation, and other forms of structured textual content.

此方法廣泛用於自然語言生成任務，包括自動化創意敘事、技術文件與各類結構化文本的撰寫。

5. Conversational Agents with State

5. 具狀態的對話式代理人

Although comprehensive state management architectures employ methods more complex than sequential linking, prompt chaining provides a foundational mechanism for preserving conversational continuity. This technique maintains context by constructing each conversational turn as a new prompt that systematically incorporates information or extracted entities from preceding interactions in the dialogue sequence.

雖然完整的狀態管理架構比序列串接更複雜，但提示串接仍提供維持對話連貫性的基礎機制。此方法將每一輪對話構造成新的提示，並系統性納入前序互動中抽取出的資訊或實體，以維持脈絡。

- Prompt 1: Process User Utterance 1, identify intent and key entities.
- Processing: Update conversation state with intent and entities.
- Prompt 2: Based on current state, generate a response and/or identify the next required piece of information.
- Repeat for subsequent turns, with each new user utterance initiating a chain that leverages the accumulating conversation history (state).
- 提示 1：處理使用者第一句話，辨識意圖與關鍵實體。
- 處理：用意圖與實體更新對話狀態。
- 提示 2：根據當前狀態生成回應，並/或找出下一步需要的資訊。
- 後續回合重複，以累積的對話歷史（狀態）啟動新的提示鏈。

This principle is fundamental to the development of conversational agents, enabling them to maintain context and coherence across extended, multi-turn dialogues. By

preserving the conversational history, the system can understand and appropriately respond to user inputs that depend on previously exchanged information.

這個原則是對話式代理人的基礎，讓系統能在長對話中維持脈絡與一致性。透過保留對話歷史，系統能理解並回應依賴先前資訊的使用者輸入。

6. Code Generation and Refinement

6. 程式碼生成與修正

The generation of functional code is typically a multi–stage process, requiring a problem to be decomposed into a sequence of discrete logical operations that are executed progressively

產生可運作的程式碼通常是多階段流程，需要把問題拆成一系列離散的邏輯操作，逐步執行。

- Prompt 1: Understand the user's request for a code function. Generate pseudocode or an outline.
- Prompt 2: Write the initial code draft based on the outline.
- Prompt 3: Identify potential errors or areas for improvement in the code (perhaps using a static analysis tool or another LLM call).
- Prompt 4: Rewrite or refine the code based on the identified issues.
- Prompt 5: Add documentation or test cases.
- 提示 1：理解使用者對功能的需求，產生偽碼或大綱。
- 提示 2：依大綱撰寫初版程式碼。
- 提示 3：找出潛在錯誤或改進處（可用靜態分析工具或另一個 LLM 呼叫）。

- 提示 4：依問題修正或優化程式碼。

- 提示 5：補上文件或測試案例。

In applications such as AI–assisted software development, the utility of prompt chaining stems from its capacity to decompose complex coding tasks into a series of manageable sub–problems. This modular structure reduces the operational complexity for the large language model at each step. Critically, this approach also allows for the insertion of deterministic logic between model calls, enabling intermediate data processing, output validation, and conditional branching within the workflow. By this method, a single, multifaceted request that could otherwise lead to unreliable or incomplete results is converted into a structured sequence of operations managed by an underlying execution framework.

在 AI 輔助軟體開發中，提示串接的價值在於能把複雜編碼任務拆成可管理的子問題。這種模組化結構降低了模型每一步的操作複雜度。更重要的是，它允許在模型呼叫之間插入確定性邏輯，支援中間資料處理、輸出驗證與條件分支。透過此方法，原本可能不可靠或不完整的單次多面向需求，會被轉換成由底層執行框架管理的結構化操作序列。

7. Multimodal and multi–step reasoning

7. 多模態與多步推理

Analyzing datasets with diverse modalities necessitates breaking down the problem into smaller, prompt–based tasks. For example, interpreting an image that contains a picture with embedded text, labels highlighting specific text segments, and tabular data explaining each label, requires such an approach.

分析多模態資料集通常需要把問題拆成較小的提示任務。例如，解析一張圖像，其中包含嵌入文字、標註文字片段的標籤，以及說明標籤的表格資料，便需要此類分解流程。

- Prompt 1: Extract and comprehend the text from the user's image request.

- Prompt 2: Link the extracted image text with its corresponding labels.

- Prompt 3: Interpret the gathered information using a table to determine the

required output.

- 提示 1：從使用者的影像請求中擷取並理解文字。
- 提示 2：把擷取的影像文字與對應標籤連結。
- 提示 3：使用表格解讀已蒐集的資訊，產出所需結果。

Hands-On Code Example

動手實作範例

Implementing prompt chaining ranges from direct, sequential function calls within a script to the utilization of specialized frameworks designed to manage control flow, state, and component integration. Frameworks such as LangChain, LangGraph, Crew AI, and the Google Agent Development Kit (ADK) offer structured environments for constructing and executing these multi-step processes, which is particularly advantageous for complex architectures.

提示串接的實作方式從腳本中的直接序列式函式呼叫，到使用專門框架來管理控制流程、狀態與元件整合皆可。LangChain、LangGraph、Crew AI 與 Google Agent Development Kit（ADK）等框架提供結構化環境以建構與執行多步驟流程，對複雜架構特別有利。

For the purpose of demonstration, LangChain and LangGraph are suitable choices as their core APIs are explicitly designed for composing chains and graphs of operations. LangChain provides foundational abstractions for linear sequences, while LangGraph extends these capabilities to support stateful and cyclical computations, which are necessary for implementing more sophisticated agentic behaviors. This example will focus on a fundamental linear sequence.

作為示範，LangChain 與 LangGraph 是合適的選擇，因為它們的核心 API 明確設計用於組合鏈與圖狀操作。LangChain 提供線性序列的基礎抽象；LangGraph 擴展為支援具狀態與循環計算，適合更進階的代理式行為。本例將聚焦於基礎的線性序列。

The following code implements a two-step prompt chain that functions as a data

processing pipeline. The initial stage is designed to parse unstructured text and extract specific information. The subsequent stage then receives this extracted output and transforms it into a structured data format.

以下程式碼實作了兩步驟提示鏈，作為資料處理管線。第一步解析非結構化文本並抽取特定資訊；第二步接收抽取結果並轉換為結構化資料格式。

To replicate this procedure, the required libraries must first be installed. This can be accomplished using the following command:

要複現此流程，需先安裝必要的套件，可使用以下指令：

```
pip install langchain langchain-community langchain-openai langgraph
```

Note that langchain-openai can be substituted with the appropriate package for a different model provider. Subsequently, the execution environment must be configured with the necessary API credentials for the selected language model provider, such as OpenAI, Google Gemini, or Anthropic.

注意：若使用不同模型供應商，可將 langchain-openai 替換為相應套件。接著需設定執行環境，填入所選模型供應商（如 OpenAI、Google Gemini 或 Anthropic）的 API 憑證。

```
import os
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

# For better security, load environment variables from a .env file
# from dotenv import load_dotenv
# load_dotenv()
# Make sure your OPENAI_API_KEY is set in the .env file

# Initialize the Language Model (using ChatOpenAI is recommended)

llm = ChatOpenAI(temperature=0)

# --- Prompt 1: Extract Information ---

prompt_extract = ChatPromptTemplate.from_template(
```

```
    "Extract the technical specifications from the following text:\n\n{text_input}"
)

# --- Prompt 2: Transform to JSON ---

prompt_transform = ChatPromptTemplate.from_template(
    "Transform the following specifications into a JSON object with 'cpu', 'memory', and
)

# --- Build the Chain using LCEL ---
# The StrOutputParser() converts the LLM's message output to a simple string.
extraction_chain = prompt_extract | llm | StrOutputParser()

# The full chain passes the output of the extraction chain into the 'specifications'
# variable for the transformation prompt.
full_chain = (
    {"specifications": extraction_chain}
        |
    prompt_transform
        |
    llm
        |
    StrOutputParser()
)

# --- Run the Chain ---

input_text = "The new laptop model features a 3.5 GHz octa-core processor, 16GB of RAM,

# Execute the chain with the input text dictionary.
final_result = full_chain.invoke({"text_input": input_text})
print("\n--- Final JSON Output ---")
print(final_result)
```

This Python code demonstrates how to use the LangChain library to process text. It

utilizes two separate prompts: one to extract technical specifications from an input string and another to format these specifications into a JSON object. The ChatOpenAI model is employed for language model interactions, and the StrOutputParser ensures the output is in a usable string format. The LangChain Expression Language (LCEL) is used to elegantly chain these prompts and the language model together. The first chain, `extraction_chain`, extracts the specifications. The `full_chain` then takes the output of the extraction and uses it as input for the transformation prompt. A sample input text describing a laptop is provided. The `full_chain` is invoked with this text, processing it through both steps. The final result, a JSON string containing the extracted and formatted specifications, is then printed.

這段 Python 程式碼示範如何使用 LangChain 進行文字處理。它使用兩個提示：第一個從輸入字串抽取技術規格，第二個將規格整理成 JSON 物件。ChatOpenAI 模型用於語言模型互動，StrOutputParser 確保輸出為可用字串格式。LangChain Expression Language（LCEL）讓提示與模型能優雅地串接。第一個鏈 `extraction_chain` 先抽取規格，`full_chain` 再把抽取結果輸入到轉換提示。範例輸入為一段描述筆電規格的文字，`full_chain` 會執行兩步驟並輸出最終 JSON 字串。

Context Engineering and Prompt Engineering

脈絡工程與提示工程

Context Engineering (see Fig.1) is the systematic discipline of designing, constructing, and delivering a complete informational environment to an AI model prior to token generation. This methodology asserts that the quality of a model's output is less dependent on the model's architecture itself and more on the richness of the context provided.

脈絡工程（見圖 1）是指在模型生成 token 前，系統性地設計、建構並提供完整資訊環境的學科。此方法認為模型輸出的品質較少取決於模型架構本身，而更取決於所提供脈絡的豐富程度。

Fig.1: Context Engineering is the discipline of building a rich, comprehensive informational environment for an AI, as the quality of this context is a primary factor in enabling advanced Agentic performance.

圖 1：脈絡工程是為 AI 建立豐富、完整資訊環境的學科，脈絡品質是驅動進階代理式表

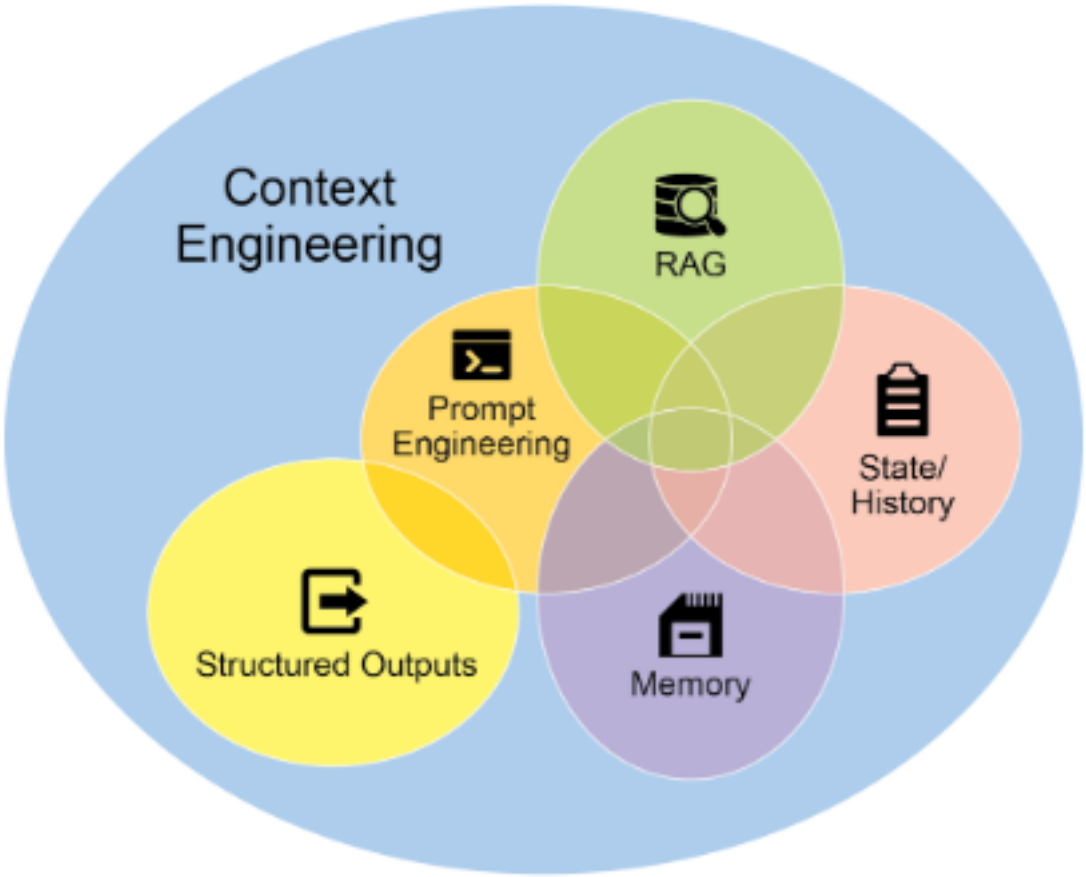

Figure 1: Context Engineering

現的主要因素之一。

It represents a significant evolution from traditional prompt engineering, which focuses primarily on optimizing the phrasing of a user's immediate query. Context Engineering expands this scope to include several layers of information, such as the **system prompt**, which is a foundational set of instructions defining the AI's operational parameters—for instance, "You are a technical writer; your tone must be formal and precise." The context is further enriched with external data. This includes retrieved documents, where the AI actively fetches information from a knowledge base to inform its response, such as pulling technical specifications for a project. It also incorporates tool outputs, which are the results from the AI using an external API to obtain real-time data, like querying a calendar to determine a user's availability. This explicit data is combined with critical implicit data, such as user identity, interaction history, and environmental state. The core principle is that even advanced models underperform when provided with a limited or poorly constructed view of the operational environment.

它是對傳統提示工程的重要演進。傳統提示工程主要在優化使用者當下問題的措辭，而脈絡工程則擴展到多層資訊，例如 **system prompt**（系統提示），即定義 AI 操作參數的基礎指令，例如 「你是技術寫作人員，語氣必須正式且精確。」脈絡也會加入外部資料，包括檢索文件（AI 從知識庫主動擷取資訊以支援回應，如提取專案技術規格）與工具輸出(AI 使用外部 API 取得即時資料，例如查詢行事曆以判斷可用時間）。這些顯性資料會與關鍵隱性資料（如使用者身分、互動歷史、環境狀態）結合。核心原則是：即使是進階模型，在資訊環境有限或建構不良時也會表現不佳。

This practice, therefore, reframes the task from merely answering a question to building a comprehensive operational picture for the agent. For example, a context-engineered agent would not just respond to a query but would first integrate the user's calendar availability (a tool output), the professional relationship with an email's recipient (implicit data), and notes from previous meetings (retrieved documents). This allows the model to generate outputs that are highly relevant, personalized, and pragmatically useful. The "engineering"component involves creating robust pipelines to fetch and transform this data at runtime and establishing feedback loops to continually improve context quality.

因此，這項實作把任務從「回答問題」重新定位為「為代理人建立完整的操作情境」。例如，一個具脈絡工程的代理人不會只回應問題，而會先整合使用者行事曆可用性（工具輸

出）、與收件人的專業關係（隱性資料），以及先前會議筆記（檢索文件）。這使模型能產生更相關、個人化且實用的輸出。「工程」成分則是建立穩健流程，於執行時擷取並轉換資料，並建立回饋迴路以持續提升脈絡品質。

To implement this, specialized tuning systems can be used to automate the improvement process at scale. For example, tools like Google's Vertex AI prompt optimizer can enhance model performance by systematically evaluating responses against a set of sample inputs and predefined evaluation metrics. This approach is effective for adapting prompts and system instructions across different models without requiring extensive manual rewriting. By providing such an optimizer with sample prompts, system instructions, and a template, it can programmatically refine the contextual inputs, offering a structured method for implementing the feedback loops required for sophisticated Context Engineering.

為了落實此方法，可使用專門的調校系統在規模上自動改進。例如 Google 的 Vertex AI prompt optimizer 可透過一組樣本輸入與預先定義的評估指標，系統性地評估回應並提升模型表現。這種方法能在不同模型之間調整提示與系統指令，避免大量手動重寫。只要提供範例提示、系統指令與模板，優化器就能程式化地精煉脈絡輸入，提供一套結構化方式來實施高階脈絡工程所需的回饋迴路。

This structured approach is what differentiates a rudimentary AI tool from a more sophisticated and contextually-aware system. It treats the context itself as a primary component, placing critical importance on what the agent knows, when it knows it, and how it uses that information. The practice ensures the model has a well-rounded understanding of the user's intent, history, and current environment. Ultimately, Context Engineering is a crucial methodology for advancing stateless chatbots into highly capable, situationally-aware systems.

這種結構化方法正是區分初階 AI 工具與更成熟、具情境感系統的關鍵。它把脈絡視為主要元件，強調代理人「知道什麼、何時知道、如何使用」的重要性。這種實作確保模型能全面理解使用者意圖、歷史與當前環境。最終，脈絡工程是把無狀態聊天機器人推進成高能力、情境感知系統的關鍵方法。

At a Glance

一覽

What: Complex tasks often overwhelm LLMs when handled within a single prompt, leading to significant performance issues. The cognitive load on the model increases the likelihood of errors such as overlooking instructions, losing context, and generating incorrect information. A monolithic prompt struggles to manage multiple constraints and sequential reasoning steps effectively. This results in unreliable and inaccurate outputs, as the LLM fails to address all facets of the multifaceted request.

是什麼：複雜任務在單一提示內處理時往往會壓垮 LLM，導致明顯效能問題。模型認知負荷增加後，容易忽略指令、失去脈絡、產生錯誤資訊。單一提示難以有效管理多重限制與序列推理步驟，結果就是不可靠且不精準的輸出，因為 LLM 無法涵蓋多面向需求的所有面向。

Why: Prompt chaining provides a standardized solution by breaking down a complex problem into a sequence of smaller, interconnected sub-tasks. Each step in the chain uses a focused prompt to perform a specific operation, significantly improving reliability and control. The output from one prompt is passed as the input to the next, creating a logical workflow that progressively builds towards the final solution. This modular, divide-and-conquer strategy makes the process more manageable, easier to debug, and allows for the integration of external tools or structured data formats between steps. This pattern is foundational for developing sophisticated, multi-step Agentic systems that can plan, reason, and execute complex workflows.

為什麼：提示串接透過把複雜問題拆成一連串相互連結的小任務，提供標準化解法。鏈中的每一步使用聚焦提示執行特定操作，大幅提升可靠性與可控性。前一步輸出成為下一步輸入，形成循序漸進的邏輯流程。此模組化、分而治之的策略讓流程更易管理、更好除錯，也能在步驟間整合外部工具或結構化資料格式。這是建構能規劃、推理並執行複雜流程的多步驟代理式系統的基礎模式。

Rule of thumb: Use this pattern when a task is too complex for a single prompt, involves multiple distinct processing stages, requires interaction with external tools between steps, or when building Agentic systems that need to perform multi-step reasoning and maintain state.

經驗法則：當任務對單一提示而言過於複雜、包含多個不同處理階段、需要在步驟間與外

部工具互動，或要建構需多步推理並維持狀態的代理式系統時，就應使用此模式。

Visual summary:

視覺摘要：

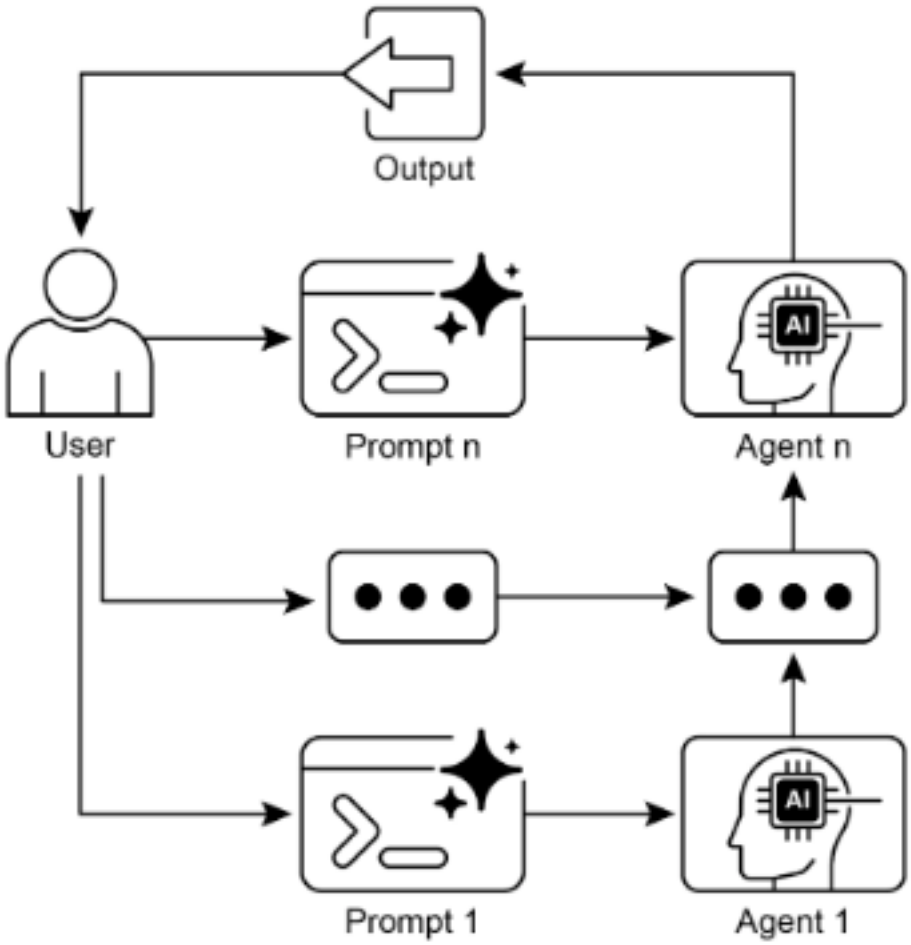

Figure 2: Prompt Chaining Pattern

Fig. 2: Prompt Chaining Pattern: Agents receive a series of prompts from the user, with the output of each agent serving as the input for the next in the chain.

圖 2：提示串接模式：代理人接收一系列提示，每個代理人的輸出作為下一個的輸入。

Key Takeaways

重點整理

Here are some key takeaways:

重點如下：

- Prompt Chaining breaks down complex tasks into a sequence of smaller, focused steps. This is occasionally known as the Pipeline pattern.
- Each step in a chain involves an LLM call or processing logic, using the output of the previous step as input.
- This pattern improves the reliability and manageability of complex interactions with language models.
- Frameworks like LangChain/LangGraph, and Google ADK provide robust tools to define, manage, and execute these multi-step sequences.
- 提示串接把複雜任務拆成一連串較小且聚焦的步驟，也常被稱為流水線模式。
- 鏈中的每一步都涉及 LLM 呼叫或處理邏輯，並以前一步輸出為輸入。
- 此模式提升與語言模型互動的可靠性與可管理性。
- LangChain/LangGraph 與 Google ADK 等框架提供強大工具以定義、管理並執行多步驟序列。

Conclusion

結論

By deconstructing complex problems into a sequence of simpler, more manageable sub-tasks, prompt chaining provides a robust framework for guiding large language

models. This “divide–and–conquer”strategy significantly enhances the reliability and control of the output by focusing the model on one specific operation at a time. As a foundational pattern, it enables the development of sophisticated AI agents capable of multi–step reasoning, tool integration, and state management. Ultimately, mastering prompt chaining is crucial for building robust, context–aware systems that can execute intricate workflows well beyond the capabilities of a single prompt.

透過把複雜問題拆成一連串更簡單、可管理的子任務，提示串接為引導大型語言模型提供了穩健的框架。這種「分而治之」策略讓模型一次只專注於單一操作，顯著提升輸出可靠性與可控性。作為基礎模式，它使得能進行多步推理、工具整合與狀態管理的高階 AI 代理人成為可能。最終，掌握提示串接是建構穩健、具脈絡感知系統的關鍵，能執行遠超單一提示能力的精密流程。

References

參考資料

1. LangChain Documentation on LCEL: https://python.langchain.com/v0.2/docs/core_modules/expression_language/
2. LangGraph Documentation: https://langchain–ai.github.io/langgraph/

3. Prompt Engineering Guide – Chaining Prompts: https://www.promptingguide.ai/techniques/chaining
4. OpenAI API Documentation (General Prompting Concepts): https://platform.openai.com/docs/guides/gpt/prompting
5. Crew AI Documentation (Tasks and Processes): https://docs.crewai.com/
6. Google AI for Developers (Prompting Guides): https://cloud.google.com/discover/what–is–prompt–engineering?hl=en
7. Vertex Prompt Optimizer https://cloud.google.com/vertex–ai/generative–ai/docs/learn/prompts/prompt–optimizer

Chapter 2: Routing

第 2 章：路由

Routing Pattern Overview

路由模式概覽

While sequential processing via prompt chaining is a foundational technique for executing deterministic, linear workflows with language models, its applicability is limited in scenarios requiring adaptive responses. Real-world agentic systems must often arbitrate between multiple potential actions based on contingent factors, such as the state of the environment, user input, or the outcome of a preceding operation. This capacity for dynamic decision-making, which governs the flow of control to different specialized functions, tools, or sub-processes, is achieved through a mechanism known as routing.

雖然透過提示串接進行的序列處理是執行確定性線性流程的基礎技術，但在需要自適應回應的情境中，其適用性有限。真實世界的代理式系統往往必須在多種可能行動之間做仲裁，依據環境狀態、使用者輸入或前一步操作結果等條件因素來決定。這種動態決策能力，用來控制流向不同的專門功能、工具或子流程，正是透過「路由」機制達成。

Routing introduces conditional logic into an agent's operational framework, enabling a shift from a fixed execution path to a model where the agent dynamically evaluates specific criteria to select from a set of possible subsequent actions. This allows for more flexible and context-aware system behavior.

路由將條件式邏輯引入代理人的作業框架，使其從固定執行路徑轉為能動態評估特定條件、從多個可能後續行動中選擇的模式。這讓系統行為更彈性、也更具脈絡感知。

For instance, an agent designed for customer inquiries, when equipped with a routing function, can first classify an incoming query to determine the user's intent. Based on this classification, it can then direct the query to a specialized agent for direct question-answering, a database retrieval tool for account information, or an escalation procedure for complex issues, rather than defaulting to a single, predetermined response pathway. Therefore, a more sophisticated agent using routing could:

例如，用於處理客戶詢問的代理人若具備路由功能，便可先對來詢分類以判斷使用者意

圖。依據分類結果，它可以把問題導向專門的問答代理人、用於查詢帳戶資訊的資料庫檢索工具，或複雜問題的升級流程，而不是固定走同一條回應路徑。因此，使用路由的進階代理人可以：

1. Analyze the user's query.

2. **Route** the query based on its intent:
    - If the intent is "check order status", route to a sub-agent or tool chain that interacts with the order database.

    - If the intent is "product information", route to a sub-agent or chain that searches the product catalog.

    - If the intent is "technical support", route to a different chain that accesses troubleshooting guides or escalates to a human.

    - If the intent is unclear, route to a clarification sub-agent or prompt chain.
3. 分析使用者問題。

4. 依 意圖 進行 **路由**：
    - 若意圖是「查詢訂單狀態」，路由到可連接訂單資料庫的子代理人或工具鏈。

    - 若意圖是「產品資訊」，路由到可搜尋產品目錄的子代理人或鏈。

    - 若意圖是「技術支援」，路由到可存取疑難排解指南或轉接人工的不同鏈。

    - 若意圖不明，路由到澄清子代理人或提示鏈。

The core component of the Routing pattern is a mechanism that performs the evaluation and directs the flow. This mechanism can be implemented in several ways:

路由模式的核心元件是負責評估並導引流程的機制。此機制可用多種方式實作：

- **LLM-based Routing:** The language model itself can be prompted to analyze the input and output a specific identifier or instruction that indicates the next step or destination. For example, a prompt might ask the LLM to "Analyze the following user query and output only the category: 'Order Status', 'Product

Info', 'Technical Support', or 'Other'."The agentic system then reads this output and directs the workflow accordingly.

- **Embedding-based Routing:** The input query can be converted into a vector embedding (see RAG, Chapter 14). This embedding is then compared to embeddings representing different routes or capabilities. The query is routed to the route whose embedding is most similar. This is useful for semantic routing, where the decision is based on the meaning of the input rather than just keywords.

- **Rule-based Routing:** This involves using predefined rules or logic (e.g., if-else statements, switch cases) based on keywords, patterns, or structured data extracted from the input. This can be faster and more deterministic than LLM-based routing, but is less flexible for handling nuanced or novel inputs.

- **Machine Learning Model-Based Routing**: it employs a discriminative model, such as a classifier, that has been specifically trained on a small corpus of labeled data to perform a routing task. While it shares conceptual similarities with embedding-based methods, its key characteristic is the supervised fine-tuning process, which adjusts the model's parameters to create a specialized routing function. This technique is distinct from LLM-based routing because the decision-making component is not a generative model executing a prompt at inference time. Instead, the routing logic is encoded within the fine-tuned model's learned weights. While LLMs may be used in a pre-processing step to generate synthetic data for augmenting the training set, they are not involved in the real-time routing decision itself.
- **LLM 路由：**可用語言模型本身分析輸入，輸出特定識別符或指令以指示下一步或目的地。例如提示 LLM：「分析以下使用者問題，只輸出分類：'Order Status'、'Product Info'、'Technical Support'或'Other'。」代理式系統讀取此輸出並導引流程。

- **Embedding 路由：**將輸入轉為向量嵌入（見第 14 章 RAG），再與代表不同路由或

能力的嵌入比較，將問題導向最相似的路由。這對語意路由很有用，決策依據語意而非關鍵字。

- **規則式路由：**以預先定義的規則或邏輯（如 if–else、switch）依據關鍵字、模式或結構化資料進行。此方法比 LLM 路由更快、更具決定性，但對細微或新穎輸入的彈性較低。

- **機器學習模型路由：**使用判別式模型（例如分類器）在一小型標註資料上訓練，用於路由任務。它與 embedding 方法概念上相似，但關鍵在於監督式微調，透過調整參數打造專用路由功能。此方法不同於 LLM 路由，因為決策元件不是在推論時執行提示的生成模型；路由邏輯被編碼在微調後的權重中。LLM 可用於前處理生成合成資料以擴充訓練集，但不參與即時路由決策。

Routing mechanisms can be implemented at multiple junctures within an agent's operational cycle. They can be applied at the outset to classify a primary task, at intermediate points within a processing chain to determine a subsequent action, or during a subroutine to select the most appropriate tool from a given set.

路由機制可在代理人作業週期的多個節點上實作。可在一開始就分類主要任務、在處理鏈的中途決定下一步行動，或在子程序中從既有工具集合中選擇最合適的工具。

Computational frameworks such as LangChain, LangGraph, and Google's Agent Developer Kit (ADK) provide explicit constructs for defining and managing such conditional logic. With its state–based graph architecture, LangGraph is particularly well–suited for complex routing scenarios where decisions are contingent upon the accumulated state of the entire system. Similarly, Google's ADK provides foundational components for structuring an agent's capabilities and interaction models, which serve as the basis for implementing routing logic. Within the execution environments provided by these frameworks, developers define the possible operational paths and the functions or model–based evaluations that dictate the transitions between nodes in the computational graph.

LangChain、LangGraph 與 Google Agent Developer Kit（ADK）等框架提供明確的結構來定義與管理這類條件邏輯。LangGraph 以狀態為基礎的圖狀架構，特別適合決策依賴整體系統累積狀態的複雜路由情境。Google ADK 也提供建構代理人能力與互動模型的基礎元件，作為實作路由邏輯的基石。在這些框架的執行環境中，開發者定義可能的操作

路徑，以及決定圖中節點轉移的函式或模型評估。

The implementation of routing enables a system to move beyond deterministic sequential processing. It facilitates the development of more adaptive execution flows that can respond dynamically and appropriately to a wider range of inputs and state changes.

實作路由讓系統超越確定性的序列處理，得以建構更具適應性的執行流程，能對更廣泛的輸入與狀態變化做出動態且適切的回應。

Practical Applications & Use Cases

實務應用與使用情境

The routing pattern is a critical control mechanism in the design of adaptive agentic systems, enabling them to dynamically alter their execution path in response to variable inputs and internal states. Its utility spans multiple domains by providing a necessary layer of conditional logic.

路由模式是設計自適應代理式系統的重要控制機制，使其能依輸入與內部狀態變化動態調整執行路徑。它在多個領域發揮作用，提供必要的條件式邏輯層。

In human–computer interaction, such as with virtual assistants or AI–driven tutors, routing is employed to interpret user intent. An initial analysis of a natural language query determines the most appropriate subsequent action, whether it is invoking a specific information retrieval tool, escalating to a human operator, or selecting the next module in a curriculum based on user performance. This allows the system to move beyond linear dialogue flows and respond contextually.

在人機互動情境中，例如虛擬助理或 AI 教學系統，路由用來解讀使用者意圖。對自然語言問題的初步分析可決定最適合的下一步動作，例如呼叫特定資訊檢索工具、升級給人工處理，或依學習表現選擇下一個課程模組。這讓系統能跳脫線性對話流程，改以情境回應。

Within automated data and document processing pipelines, routing serves as a classification and distribution function. Incoming data, such as emails, support tickets, or API payloads, is analyzed based on content, metadata, or format. The system then directs each item to a corresponding workflow, such as a sales lead ingestion pro–

cess, a specific data transformation function for JSON or CSV formats, or an urgent issue escalation path.

在自動化資料與文件處理管線中，路由扮演分類與分派功能。系統會依內容、Metadata 或格式分析進件資料（如郵件、支援工單、API payload），再將每個項目導向相對應的流程，例如銷售線索匯入、特定 JSON/CSV 轉換流程，或緊急問題升級路徑。

In complex systems involving multiple specialized tools or agents, routing acts as a high-level dispatcher. A research system composed of distinct agents for searching, summarizing, and analyzing information would use a router to assign tasks to the most suitable agent based on the current objective. Similarly, an AI coding assistant uses routing to identify the programming language and user's intent—to debug, explain, or translate—before passing a code snippet to the correct specialized tool.

在包含多個專門工具或代理人的複雜系統中，路由作為高階分派器。由搜尋、摘要、分析代理人組成的研究系統，會用路由器根據當前目標指派任務給最合適的代理人。同樣地，AI 程式碼助理會先判斷程式語言與使用者意圖（除錯、解釋或翻譯），再把程式碼片段送到正確的專門工具。

Ultimately, routing provides the capacity for logical arbitration that is essential for creating functionally diverse and context-aware systems. It transforms an agent from a static executor of pre-defined sequences into a dynamic system that can make decisions about the most effective method for accomplishing a task under changing conditions.

最終，路由提供必要的邏輯仲裁能力，使系統能具備多樣功能並具情境感知。它把代理人從固定序列的執行者，轉變為能在變動條件下選擇最有效方法完成任務的動態系統。

Hands-On Code Example (LangChain)

動手實作範例（LangChain）

Implementing routing in code involves defining the possible paths and the logic that decides which path to take. Frameworks like LangChain and LangGraph provide specific components and structures for this. LangGraph's state-based graph structure is particularly intuitive for visualizing and implementing routing logic.

在程式中實作路由，需要定義可能的路徑與決定走哪條路的邏輯。LangChain 與 Lang-

Graph 等框架提供了對應的元件與結構。LangGraph 以狀態為基礎的圖狀結構，特別直覺，適合視覺化並實作路由邏輯。

This code demonstrates a simple agent-like system using LangChain and Google's Generative AI. It sets up a "coordinator"that routes user requests to different simulated "sub-agent"handlers based on the request's intent (booking, information, or unclear). The system uses a language model to classify the request and then delegates it to the appropriate handler function, simulating a basic delegation pattern often seen in multi-agent architectures.

以下程式碼示範使用 LangChain 與 Google Generative AI 建立簡單的代理式系統。它設置一個「協調者」依據請求意圖（訂位、資訊、或不明）把使用者請求路由到不同的「子代理人」處理器。系統用語言模型分類請求，接著委派給適當的處理函式，模擬多代理人架構中常見的基本委派模式。

First, ensure you have the necessary libraries installed:

首先，請確認已安裝必要套件：

```
pip install langchain langgraph google-cloud-aiplatform langchain-google-genai google-ad
```

You will also need to set up your environment with your API key for the language model you choose (e.g., OpenAI, Google Gemini, Anthropic).

你也需要設定所選模型供應商（如 OpenAI、Google Gemini 或 Anthropic）的 API 金鑰。

```
# Copyright (c) 2025 Marco Fago
# https://www.linkedin.com/in/marco-fago/
#
# This code is licensed under the MIT License.
# See the LICENSE file in the repository for the full license text.

from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough, RunnableBranch


# --- Configuration ---
```

```python
# Ensure your API key environment variable is set (e.g., GOOGLE_API_KEY)
try:
    llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0)
    print(f"Language model initialized: {llm.model}")
except Exception as e:
    print(f"Error initializing language model: {e}")
    llm = None


# --- Define Simulated Sub-Agent Handlers (equivalent to ADK sub_agents) ---
def booking_handler(request: str) -> str:
    """Simulates the Booking Agent handling a request."""
    print("\n--- DELEGATING TO BOOKING HANDLER ---")
    return f"Booking Handler processed request: '{request}'. Result: Simulated booking a


def info_handler(request: str) -> str:
    """Simulates the Info Agent handling a request."""
    print("\n--- DELEGATING TO INFO HANDLER ---")
    return f"Info Handler processed request: '{request}'. Result: Simulated information


def unclear_handler(request: str) -> str:
    """Handles requests that couldn't be delegated."""
    print("\n--- HANDLING UNCLEAR REQUEST ---")
    return f"Coordinator could not delegate request: '{request}'. Please clarify."


# --- Define Coordinator Router Chain (equivalent to ADK coordinator's instruction) --
# This chain decides which handler to delegate to.
coordinator_router_prompt = ChatPromptTemplate.from_messages([
    (
        "system",
        """Analyze the user's request and determine which specialist handler should proc
```

```
        - If the request is related to booking flights or hotels,
          output 'booker'.
        - For all other general information questions, output 'info'.
        - If the request is unclear or doesn't fit either category,
          output 'unclear'.
        ONLY output one word: 'booker', 'info', or 'unclear'."""
    ),
    ("user", "{request}")
])

if llm:
    coordinator_router_chain = coordinator_router_prompt | llm | StrOutputParser()


# --- Define the Delegation Logic (equivalent to ADK's Auto-Flow based on sub_agents)
# Use RunnableBranch to route based on the router chain's output.

# Define the branches for the RunnableBranch
branches = {
    "booker": RunnablePassthrough.assign(
        output=lambda x: booking_handler(x['request']['request'])
    ),
    "info": RunnablePassthrough.assign(
        output=lambda x: info_handler(x['request']['request'])
    ),
    "unclear": RunnablePassthrough.assign(
        output=lambda x: unclear_handler(x['request']['request'])
    ),
}

# Create the RunnableBranch. It takes the output of the router chain
# and routes the original input ('request') to the corresponding handler.
delegation_branch = RunnableBranch(
    (lambda x: x['decision'].strip() == 'booker', branches["booker"]),  # Added .strip(
```

```
    (lambda x: x['decision'].strip() == 'info', branches["info"]),      # Added .strip()
    branches["unclear"]  # Default branch for 'unclear' or any other output
)

# Combine the router chain and the delegation branch into a single runnable
# The router chain's output ('decision') is passed along with the original input ('req
# to the delegation_branch.
coordinator_agent = {
    "decision": coordinator_router_chain,
    "request": RunnablePassthrough()
} | delegation_branch | (lambda x: x['output'])  # Extract the final output


# --- Example Usage ---
def main():
    if not llm:
        print("\nSkipping execution due to LLM initialization failure.")
        return

    print("--- Running with a booking request ---")
    request_a = "Book me a flight to London."
    result_a = coordinator_agent.invoke({"request": request_a})
    print(f"Final Result A: {result_a}")

    print("\n--- Running with an info request ---")
    request_b = "What is the capital of Italy?"
    result_b = coordinator_agent.invoke({"request": request_b})
    print(f"Final Result B: {result_b}")

    print("\n--- Running with an unclear request ---")
    request_c = "Tell me about quantum physics."
    result_c = coordinator_agent.invoke({"request": request_c})
    print(f"Final Result C: {result_c}")
```

```
if __name__ == "__main__":
    main()
```

As mentioned, this Python code constructs a simple agent-like system using the LangChain library and Google's Generative AI model, specifically gemini-2.5-flash. In detail, It defines three simulated sub-agent handlers: `booking_handler`, `info_handler`, and `unclear_handler`, each designed to process specific types of requests.

如上所述，此 Python 程式碼使用 LangChain 與 Google Generative AI 模型（gemini-2.5-flash）建構簡單代理式系統。細節上，它定義了三個模擬子代理人處理器：`booking_handler`、`info_handler` 與 `unclear_handler`，各自處理特定類型的請求。

A core component is the `coordinator_router_chain`, which utilizes a ChatPrompt-Template to instruct the language model to categorize incoming user requests into one of three categories: `booker`, `info`, or `unclear`. The output of this router chain is then used by a RunnableBranch to delegate the original request to the corresponding handler function. The RunnableBranch checks the decision from the language model and directs the request data to either the `booking_handler`, `info_handler`, or `unclear_handler`. The `coordinator_agent` combines these components, first routing the request for a decision and then passing the request to the chosen handler. The final output is extracted from the handler's response.

核心元件是 `coordinator_router_chain`，它使用 ChatPromptTemplate 指示語言模型把使用者請求分類為 `booker`、`info` 或 `unclear`。路由器鏈的輸出再由 RunnableBranch 來委派原始請求給對應的處理函式。RunnableBranch 會檢查模型決策並將請求資料導向 `booking_handler`、`info_handler` 或 `unclear_handler`。`coordinator_agent` 組合這些元件，先做路由判斷，再將請求交給被選擇的處理器，最終輸出取自處理器回應。

The main function demonstrates the system's usage with three example requests, showcasing how different inputs are routed and processed by the simulated agents. Error handling for language model initialization is included to ensure robustness. The code structure mimics a basic multi-agent framework where a central coordinator delegates tasks to specialized agents based on intent.

主程式用三個範例請求展示系統用法，呈現不同輸入如何被路由並由模擬代理人處理。也加入語言模型初始化的錯誤處理以提升穩健性。整體程式結構模擬基本多代理人框架：由中央協調者依意圖委派任務給專門代理人。

Hands–On Code Example (Google ADK)

動手實作範例（Google ADK）

The Agent Development Kit (ADK) is a framework for engineering agentic systems, providing a structured environment for defining an agent's capabilities and behaviours. In contrast to architectures based on explicit computational graphs, routing within the ADK paradigm is typically implemented by defining a discrete set of "tools"that represent the agent's functions. The selection of the appropriate tool in response to a user query is managed by the framework's internal logic, which leverages an underlying model to match user intent to the correct functional handler.

Agent Development Kit（ADK）是用於建構代理式系統的框架，提供結構化環境以定義代理人能力與行為。相較於明確的計算圖架構，ADK 的路由通常透過定義一組離散的「工具」來實作，這些工具代表代理人的功能。適當工具的選擇由框架內部邏輯管理，透過底層模型將使用者意圖匹配到正確的功能處理器。

This Python code demonstrates an example of an Agent Development Kit (ADK) application using Google's ADK library. It sets up a "Coordinator"agent that routes user requests to specialized sub–agents ("Booker"for bookings and "Info"for general information) based on defined instructions. The sub–agents then use specific tools to simulate handling the requests, showcasing a basic delegation pattern within an agent system

這段 Python 程式碼示範使用 Google ADK 建立 ADK 應用。它設定一個「Coordinator」代理人，依照指令把使用者請求路由到專門子代理人（訂位用的「Booker」、一般資訊用的「Info」）。子代理人再使用特定工具模擬處理請求，展示代理系統中的基本委派模式。

```
# Copyright (c) 2025 Marco Fago
#
# This code is licensed under the MIT License.
# See the LICENSE file in the repository for the full license text.

import uuid
from typing import Dict, Any, Optional

from google.adk.agents import Agent
```

```python
from google.adk.runners import InMemoryRunner
from google.adk.tools import FunctionTool
from google.genai import types
from google.adk.events import Event


# --- Define Tool Functions ---
# These functions simulate the actions of the specialist agents.
def booking_handler(request: str) -> str:
    """
    Handles booking requests for flights and hotels.

    Args:
        request: The user's request for a booking.

    Returns:
        A confirmation message that the booking was handled.
    """
    print("-------------------------- Booking Handler Called ----------------------------
    return f"Booking action for '{request}' has been simulated."


def info_handler(request: str) -> str:
    """
    Handles general information requests.

    Args:
        request: The user's question.

    Returns:
        A message indicating the information request was handled.
    """
    print("-------------------------- Info Handler Called ----------------------------")
    return f"Information request for '{request}'. Result: Simulated information retrieva
```

```python
def unclear_handler(request: str) -> str:
    """Handles requests that couldn't be delegated."""
    return f"Coordinator could not delegate request: '{request}'. Please clarify."


# --- Create Tools from Functions ---
booking_tool = FunctionTool(booking_handler)
info_tool = FunctionTool(info_handler)

# Define specialized sub-agents equipped with their respective tools
booking_agent = Agent(
    name="Booker",
    model="gemini-2.0-flash",
    description="A specialized agent that handles all flight "
                "and hotel booking requests by calling the booking tool.",
    tools=[booking_tool],
)

info_agent = Agent(
    name="Info",
    model="gemini-2.0-flash",
    description="A specialized agent that provides general information "
                "and answers user questions by calling the info tool.",
    tools=[info_tool],
)

# Define the parent agent with explicit delegation instructions
coordinator = Agent(
    name="Coordinator",
    model="gemini-2.0-flash",
    instruction=(
        "You are the main coordinator. Your only task is to analyze "
```

```
        "incoming user requests "
        "and delegate them to the appropriate specialist agent. Do not try to answer the
        "- For any requests related to booking flights or hotels, delegate to the 'Booke
        "- For all other general information questions, delegate to the 'Info' agent."
    ),
    description="A coordinator that routes user requests to the correct specialist agent
    # The presence of sub_agents enables LLM-driven delegation (Auto-Flow) by default.
    sub_agents=[booking_agent, info_agent],
)


# --- Execution Logic ---
async def run_coordinator(runner: InMemoryRunner, request: str):
    """Runs the coordinator agent with a given request and delegates."""
    print(f"\n--- Running Coordinator with request: '{request}' ---")
    final_result = ""
    try:
        user_id = "user_123"
        session_id = str(uuid.uuid4())

        await runner.session_service.create_session(
            app_name=runner.app_name,
            user_id=user_id,
            session_id=session_id,
        )

        for event in runner.run(
            user_id=user_id,
            session_id=session_id,
            new_message=types.Content(
                role='user',
                parts=[types.Part(text=request)],
            ),
        ):
```

```
            if event.is_final_response() and event.content:
                # Try to get text directly from event.content to avoid iterating parts
                if hasattr(event.content, 'text') and event.content.text:
                    final_result = event.content.text
                elif event.content.parts:
                    # Fallback: Iterate through parts and extract text (might trigger u
                    text_parts = [part.text for part in event.content.parts if getattr(p
                    final_result = "".join(text_parts)
                # Assuming the loop should break after the final response
                break

        print(f"Coordinator Final Response: {final_result}")
        return final_result

    except Exception as e:
        print(f"An error occurred while processing your request: {e}")
        return f"An error occurred while processing your request: {e}"


async def main():
    """Main function to run the ADK example."""
    print("--- Google ADK Routing Example (ADK Auto-Flow Style) ---")
    print("Note: This requires Google ADK installed and authenticated.")

    runner = InMemoryRunner(coordinator)

    # Example Usage
    result_a = await run_coordinator(runner, "Book me a hotel in Paris.")
    print(f"Final Output A: {result_a}")

    result_b = await run_coordinator(runner, "What is the highest mountain in the world?
    print(f"Final Output B: {result_b}")

    result_c = await run_coordinator(runner, "Tell me a random fact.")  # Should go to
```

```
    print(f"Final Output C: {result_c}")

    result_d = await run_coordinator(runner, "Find flights to Tokyo next month.")  # Sho
    print(f"Final Output D: {result_d}")


if __name__ == "__main__":
    import nest_asyncio

    nest_asyncio.apply()
    await main()
```

This script consists of a main Coordinator agent and two specialized `sub_agents`: Booker and Info. Each specialized agent is equipped with a FunctionTool that wraps a Python function simulating an action. The `booking_handler` function simulates handling flight and hotel bookings, while the `info_handler` function simulates retrieving general information. The `unclear_handler` is included as a fallback for requests the coordinator cannot delegate, although the current coordinator logic doesn't explicitly use it for delegation failure in the main `run_coordinator` function.

這個腳本包含一個主要的 Coordinator 代理人與兩個專門 `sub_agents`：Booker 與 Info。每個專門代理人都配有一個 FunctionTool，包裝了模擬行動的 Python 函式。`booking_handler` 用於模擬處理機票與飯店訂位，`info_handler` 用於模擬一般資訊查詢。`unclear_handler` 作為協調者無法委派時的備援，但在 `run_coordinator` 主流程中目前沒有明確使用於委派失敗情境。

The Coordinator agent's primary role, as defined in its instruction, is to analyze incoming user messages and delegate them to either the Booker or Info agent. This delegation is handled automatically by the ADK's Auto-Flow mechanism because the Coordinator has `sub_agents` defined. The `run_coordinator` function sets up an InMemoryRunner, creates a user and session ID, and then uses the runner to process the user's request through the coordinator agent. The runner.run method processes the request and yields events, and the code extracts the final response text from the event.content.

Coordinator 代理人的主要角色（由指令定義）是分析使用者訊息並委派給 Booker 或 Info。由於 Coordinator 定義了 `sub_agents`，ADK 的 Auto-Flow 會自動處理此委派。

`run_coordinator` 會建立 InMemoryRunner、建立使用者與 session ID，並透過 runner 讓協調者處理請求。runner.run 會處理請求並產生事件，程式再從 event.content 擷取最終回應文字。

The main function demonstrates the system's usage by running the coordinator with different requests, showcasing how it delegates booking requests to the Booker and information requests to the Info agent.

主程式以不同請求展示系統用法，呈現它如何把訂位請求委派給 Booker、資訊請求委派給 Info。

At a Glance

一覽

What: Agentic systems must often respond to a wide variety of inputs and situations that cannot be handled by a single, linear process. A simple sequential workflow lacks the ability to make decisions based on context. Without a mechanism to choose the correct tool or sub-process for a specific task, the system remains rigid and non-adaptive. This limitation makes it difficult to build sophisticated applications that can manage the complexity and variability of real-world user requests.

是什麼：代理式系統常需要回應多樣化輸入與情境，這些都無法靠單一線性流程處理。簡單的序列工作流程缺乏依脈絡決策的能力。若沒有機制為特定任務選擇正確工具或子流程，系統就會僵化且缺乏適應性，難以處理真實世界使用者需求的複雜性與變異性。

Why: The Routing pattern provides a standardized solution by introducing conditional logic into an agent's operational framework. It enables the system to first analyze an incoming query to determine its intent or nature. Based on this analysis, the agent dynamically directs the flow of control to the most appropriate specialized tool, function, or sub-agent. This decision can be driven by various methods, including prompting LLMs, applying predefined rules, or using embedding-based semantic similarity. Ultimately, routing transforms a static, predetermined execution path into a flexible and context-aware workflow capable of selecting the best possible action.

為什麼：路由模式透過在代理人作業框架中導入條件式邏輯，提供標準化解法。系統先分析來詢以判斷其意圖或性質，再動態導引控制流程至最合適的專門工具、函式或子代理人。這項決策可由多種方法驅動，包括 LLM 提示、預先規則或 embedding 的語意相似

度。最終，路由把靜態預設的執行路徑轉換為彈性且具脈絡感知的流程，能選擇最佳行動。

Rule of Thumb: Use the Routing pattern when an agent must decide between multiple distinct workflows, tools, or sub-agents based on the user's input or the current state. It is essential for applications that need to triage or classify incoming requests to handle different types of tasks, such as a customer support bot distinguishing between sales inquiries, technical support, and account management questions.

經驗法則：當代理人需要依使用者輸入或當前狀態在多個不同流程、工具或子代理人之間做決策時，就應使用路由模式。對需要分類或分診來處理不同任務類型的應用（如客服機器人區分銷售詢問、技術支援與帳戶管理）尤其重要。

Visual Summary:

視覺摘要：

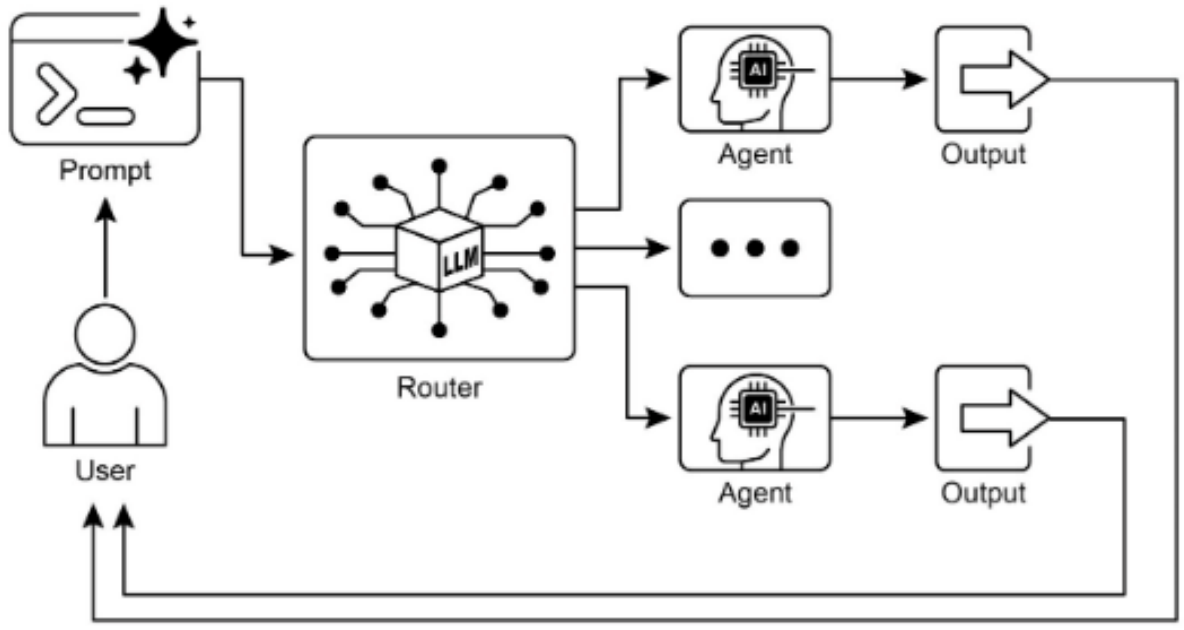

Figure 3: Router Pattern, using LLM as a Router

Fig.1: Router pattern, using an LLM as a Router

圖 1：以 LLM 作為路由器的路由模式

Key Takeaways

重點整理

- Routing enables agents to make dynamic decisions about the next step in a workflow based on conditions.
- It allows agents to handle diverse inputs and adapt their behavior, moving beyond linear execution.
- Routing logic can be implemented using LLMs, rule-based systems, or embedding similarity.
- Frameworks like LangGraph and Google ADK provide structured ways to define and manage routing within agent workflows, albeit with different architectural approaches.
- 路由讓代理人能依條件動態決定工作流程的下一步。
- 它讓代理人能處理多樣輸入並調整行為，超越線性執行。
- 路由邏輯可由 LLM、規則式系統或 embedding 相似度實作。
- LangGraph 與 Google ADK 提供結構化方式來定義與管理代理流程中的路由，但其架構取向不同。

Conclusion

結論

The Routing pattern is a critical step in building truly dynamic and responsive agentic systems. By implementing routing, we move beyond simple, linear execution flows

and empower our agents to make intelligent decisions about how to process information, respond to user input, and utilize available tools or sub-agents.

路由模式是建構真正動態且可回應的代理式系統之關鍵一步。透過實作路由，我們得以超越簡單線性流程，讓代理人能智慧地決定如何處理資訊、回應使用者輸入，並運用可用工具或子代理人。

We've seen how routing can be applied in various domains, from customer service chatbots to complex data processing pipelines. The ability to analyze input and conditionally direct the workflow is fundamental to creating agents that can handle the inherent variability of real-world tasks.

我們也看到路由可應用於多種領域，從客服聊天機器人到複雜的資料處理管線。分析輸入並以條件導向流程的能力，是打造能處理真實任務變異性的代理人的基礎。

The code examples using LangChain and Google ADK demonstrate two different, yet effective, approaches to implementing routing. LangGraph's graph-based structure provides a visual and explicit way to define states and transitions, making it ideal for complex, multi-step workflows with intricate routing logic. Google ADK, on the other hand, often focuses on defining distinct capabilities (Tools) and relies on the framework's ability to route user requests to the appropriate tool handler, which can be simpler for agents with a well-defined set of discrete actions.

LangChain 與 Google ADK 的程式碼範例展示了兩種不同但有效的路由實作方式。LangGraph 的圖狀結構提供可視化且明確的狀態與轉移定義方式，適合含複雜路由邏輯的多步驟流程。Google ADK 則偏向定義離散能力（工具），並仰賴框架將使用者請求路由到適當的工具處理器，對於動作集合明確的代理人而言更簡單。

Mastering the Routing pattern is essential for building agents that can intelligently navigate different scenarios and provide tailored responses or actions based on context. It's a key component in creating versatile and robust agentic applications.

掌握路由模式是建立能在不同情境中智慧導向並根據脈絡提供量身回應或行動的代理人的關鍵。它是打造多功能且穩健的代理式應用的重要元件。

References

參考資料

1. LangGraph Documentation: https://www.langchain.com/
2. Google Agent Developer Kit Documentation: https://google.github.io/adk-docs/

Chapter 3: Parallelization

第 3 章：平行化

Parallelization Pattern Overview

平行化模式概覽

In the previous chapters, we've explored Prompt Chaining for sequential workflows and Routing for dynamic decision-making and transitions between different paths. While these patterns are essential, many complex agentic tasks involve multiple sub-tasks that can be executed simultaneously rather than one after another. This is where the **Parallelization** pattern becomes crucial.

前兩章我們探討了用於序列流程的提示串接，以及用於動態決策與路徑切換的路由。這些模式固然重要，但許多複雜代理式任務包含多個可同時執行的子任務，而非逐一執行。此時 **平行化** 模式就成為關鍵。

Parallelization involves executing multiple components, such as LLM calls, tool usages, or even entire sub-agents, concurrently (see Fig.1). Instead of waiting for one step to complete before starting the next, parallel execution allows independent tasks to run at the same time, significantly reducing the overall execution time for tasks that can be broken down into independent parts.

平行化是指同時執行多個元件，例如 LLM 呼叫、工具使用，甚至整個子代理人（見圖 1）。不必等一個步驟完成才開始下一步，平行執行可讓彼此獨立的任務同時進行，顯著縮短可拆解為獨立部分之任務的總執行時間。

Consider an agent designed to research a topic and summarize its findings. A sequential approach might:

假設有一個代理人要研究某主題並摘要結果，若採序列做法可能是：

1. Search for Source A.
2. Summarize Source A.
3. Search for Source B.
4. Summarize Source B.
5. Synthesize a final answer from summaries A and B.
6. 搜尋來源 A。
7. 摘要來源 A。
8. 搜尋來源 B。
9. 摘要來源 B。
10. 綜合 A 與 B 的摘要得到最終答案。

A parallel approach could instead:

若採平行做法則可能是：

1. Search for Source A and Search for Source B simultaneously.
2. Once both searches are complete, Summarize Source A and Summarize Source B simultaneously.

3. Synthesize a final answer from summaries A and B (this step is typically sequential, waiting for the parallel steps to finish).
4. 同時搜尋來源 A 與來源 B。
5. 兩個搜尋完成後，同時摘要來源 A 與來源 B。
6. 綜合 A 與 B 的摘要（此步通常仍為序列，需等待平行步驟完成）。

The core idea is to identify parts of the workflow that do not depend on the output of other parts and execute them in parallel. This is particularly effective when dealing with external services (like APIs or databases) that have latency, as you can issue multiple requests concurrently.

核心概念是找出不依賴其他輸出的流程部分，將它們平行執行。這在面對具延遲的外部服務（如 API 或資料庫）時特別有效，因為可以同時送出多個請求。

Implementing parallelization often requires frameworks that support asynchronous execution or multi-threading/multi-processing. Modern agentic frameworks are designed with asynchronous operations in mind, allowing you to easily define steps that can run in parallel.

實作平行化通常需要支援非同步或多執行緒/多進程的框架。現代代理式框架多以非同步為設計核心，讓你能輕鬆定義可平行執行的步驟。

Fig.1. Example of parallelization with sub-agents

圖 1：子代理人平行化示例

Frameworks like LangChain, LangGraph, and Google ADK provide mechanisms for parallel execution. In LangChain Expression Language (LCEL), you can achieve parallel execution by combining runnable objects using operators like | (for sequential) and by structuring your chains or graphs to have branches that execute concurrently. LangGraph, with its graph structure, allows you to define multiple nodes that can be executed from a single state transition, effectively enabling parallel branches in the workflow. Google ADK provides robust, native mechanisms to facilitate and manage the parallel execution of agents, significantly enhancing the efficiency and scalability of complex, multi-agent systems. This inherent capability within the ADK framework allows developers to design and implement solutions where multiple agents can

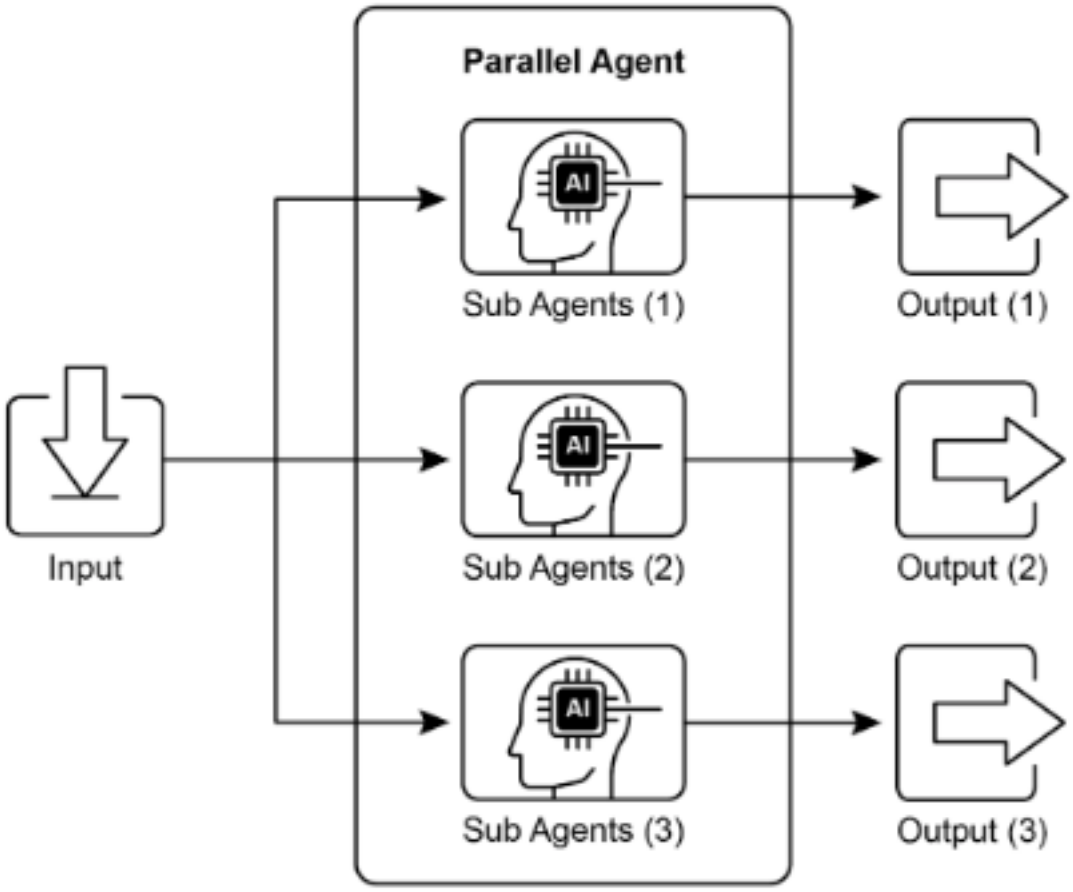

Figure 4: Parallelization with Sub-Agents

operate concurrently, rather than sequentially.

LangChain、LangGraph 與 Google ADK 等框架都提供平行執行機制。在 LangChain Expression Language（LCEL）中，可透過組合 runnable 物件達成平行執行；| 用於序列，並透過分支結構讓多個路徑同時執行。LangGraph 的圖結構允許從同一個狀態轉移啟動多個節點，實際上形成平行分支。Google ADK 則提供強健的原生機制來促進並管理代理人的平行執行，顯著提升複雜多代理人系統的效率與可擴展性。ADK 框架內建的能力讓開發者可設計讓多個代理人同時運作的方案，而非依序運作。

The Parallelization pattern is vital for improving the efficiency and responsiveness of agentic systems, especially when dealing with tasks that involve multiple independent lookups, computations, or interactions with external services. It's a key technique for optimizing the performance of complex agent workflows.

平行化模式對提升代理式系統的效率與回應性至關重要，特別是在涉及多個獨立查詢、計算或與外部服務互動的任務中。它是優化複雜代理流程效能的關鍵技術。

Practical Applications & Use Cases

實務應用與使用情境

Parallelization is a powerful pattern for optimizing agent performance across various applications:

平行化是提升代理效能的強大模式，適用於各種應用情境：

1. Information Gathering and Research

1. 資訊蒐集與研究

Collecting information from multiple sources simultaneously is a classic use case.

同時從多來源蒐集資訊是經典用例。

- **Use Case:** An agent researching a company.
  - **Parallel Tasks:** Search news articles, pull stock data, check social media mentions, and query a company database, all at the same time.

- **Benefit:** Gathers a comprehensive view much faster than sequential lookups.

- **使用情境：**代理人研究公司。
  - **平行任務：**同時搜尋新聞、拉取股價資料、檢查社群提及、查詢公司資料庫。
  - **效益：**比序列查詢更快取得全面觀點。

2. Data Processing and Analysis

2. 資料處理與分析

Applying different analysis techniques or processing different data segments concurrently.

同時套用不同分析方法或處理不同資料區段。

- **Use Case:** An agent analyzing customer feedback.
  - **Parallel Tasks:** Run sentiment analysis, extract keywords, categorize feedback, and identify urgent issues simultaneously across a batch of feedback entries.
  - **Benefit:** Provides a multi-faceted analysis quickly.
- **使用情境：**代理人分析客戶回饋。
  - **平行任務：**同時進行情緒分析、關鍵字抽取、回饋分類、急迫問題識別，並可在一批回饋上並行。
  - **效益：**快速提供多面向分析。

3. Multi-API or Tool Interaction

3. 多 API / 工具互動

Calling multiple independent APIs or tools to gather different types of information or perform different actions.

呼叫多個獨立 API 或工具以取得不同資訊或執行不同動作。

- **Use Case:** A travel planning agent.

    - **Parallel Tasks:** Check flight prices, search for hotel availability, look up local events, and find restaurant recommendations concurrently.
    - **Benefit:** Presents a complete travel plan faster.
- **使用情境：**旅遊規劃代理人。
    - **平行任務：**同時查詢機票價格、飯店空房、本地活動與餐廳推薦。
    - **效益：**更快速提供完整旅遊規劃。

4. Content Generation with Multiple Components

4. 多元組件的內容生成

Generating different parts of a complex piece of content in parallel.

將複雜內容的不同部分平行生成。

- **Use Case:** An agent creating a marketing email.
    - **Parallel Tasks:** Generate a subject line, draft the email body, find a relevant image, and create a call-to-action button text simultaneously.
    - **Benefit:** Assembles the final email more efficiently.
- **使用情境：**代理人撰寫行銷郵件。
    - **平行任務：**同時產生主旨、撰寫內文、尋找相關圖片、產生 CTA 按鈕文字。
    - **效益：**更有效率組成最終郵件。

5. Validation and Verification

5. 驗證與確認

Performing multiple independent checks or validations concurrently.

同時進行多個獨立檢查或驗證。

- **Use Case:** An agent verifying user input.
    - **Parallel Tasks:** Check email format, validate phone number, verify address against a database, and check for profanity simultaneously.

  - **Benefit:** Provides faster feedback on input validity.
- **使用情境：**代理人驗證使用者輸入。
  - **平行任務：**同時檢查 email 格式、驗證電話號碼、比對地址資料庫、檢查不雅詞彙。
  - **效益：**更快回饋輸入是否有效。

6. Multi-Modal Processing

6. 多模態處理

Processing different modalities (text, image, audio) of the same input concurrently.

同時處理同一輸入的不同模態（文字、影像、音訊）。

- **Use Case:** An agent analyzing a social media post with text and an image.
  - **Parallel Tasks:** Analyze the text for sentiment and keywords and analyze the image for objects and scene description simultaneously.
  - **Benefit:** Integrates insights from different modalities more quickly.
- **使用情境：**代理人分析含文字與圖片的社群貼文。
  - **平行任務：**同時分析文字的情緒與關鍵字、分析圖片的物件與場景。
  - **效益：**更快速整合多模態洞見。

7. A/B Testing or Multiple Options Generation

7. A/B 測試或多版本生成

Generating multiple variations of a response or output in parallel to select the best one.

平行產生多個版本以挑選最佳結果。

- **Use Case:** An agent generating different creative text options.
  - **Parallel Tasks:** Generate three different headlines for an article simultaneously using slightly different prompts or models.

    - **Benefit:** Allows for quick comparison and selection of the best option.
- **使用情境：**代理人產生多個創意文案。
    - **平行任務：**使用不同提示或模型同時產生三個標題。
    - **效益：**快速比較並選出最佳選項。

Parallelization is a fundamental optimization technique in agentic design, allowing developers to build more performant and responsive applications by leveraging concurrent execution for independent tasks.

平行化是代理式設計中的基本效能優化技術，透過獨立任務的併發執行，讓開發者打造更高效、更具回應性的應用。

Hands-On Code Example (LangChain)

動手實作範例（LangChain）

Parallel execution within the LangChain framework is facilitated by the LangChain Expression Language (LCEL). The primary method involves structuring multiple runnable components within a dictionary or list construct. When this collection is passed as input to a subsequent component in the chain, the LCEL runtime executes the contained runnables concurrently.

在 LangChain 中的平行執行由 LangChain Expression Language（LCEL）提供支援。主要做法是把多個 runnable 元件組成字典或清單結構。當這個集合傳入鏈中的下一個元件時，LCEL 會同時執行其中的 runnable。

In the context of LangGraph, this principle is applied to the graph's topology. Parallel workflows are defined by architecting the graph such that multiple nodes, lacking direct sequential dependencies, can be initiated from a single common node. These parallel pathways execute independently before their results can be aggregated at a subsequent convergence point in the graph.

在 LangGraph 中，這個原理會反映在圖的拓樸上。透過設計圖結構，使多個沒有直接序列依賴的節點能由同一節點啟動，即可形成平行流程。這些平行路徑獨立執行，之後再在圖中的匯合點聚合結果。

The following implementation demonstrates a parallel processing workflow constructed with the LangChain framework. This workflow is designed to execute two independent operations concurrently in response to a single user query. These parallel processes are instantiated as distinct chains or functions, and their respective outputs are subsequently aggregated into a unified result.

以下實作示範使用 LangChain 建構平行處理流程。此流程會針對單一使用者查詢，同時執行多個獨立操作。這些平行流程以不同鏈或函式實例化，並將各自輸出彙整成單一結果。

The prerequisites for this implementation include the installation of the requisite Python packages, such as langchain, langchain-community, and a model provider library like langchain-openai. Furthermore, a valid API key for the chosen language model must be configured in the local environment for authentication.

此實作的前置條件包含安裝必要 Python 套件，例如 langchain、langchain-community，以及模型供應商套件（如 langchain-openai）。此外需在本機環境設定所選語言模型的有效 API 金鑰以完成驗證。

```
import os
import asyncio
from typing import Optional

from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import Runnable, RunnableParallel, RunnablePassthrough


# --- Configuration ---
# Ensure your API key environment variable is set (e.g., OPENAI_API_KEY)
try:
    llm: Optional[ChatOpenAI] = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)
except Exception as e:
    print(f"Error initializing language model: {e}")
    llm = None
```

```
# --- Define Independent Chains ---
# These three chains represent distinct tasks that can be executed in parallel.
summarize_chain: Runnable = (
    ChatPromptTemplate.from_messages([
        ("system", "Summarize the following topic concisely:"),
        ("user", "{topic}"),
    ])
    | llm
    | StrOutputParser()
)

questions_chain: Runnable = (
    ChatPromptTemplate.from_messages([
        ("system", "Generate three interesting questions about the following topic:"),
        ("user", "{topic}"),
    ])
    | llm
    | StrOutputParser()
)

terms_chain: Runnable = (
    ChatPromptTemplate.from_messages([
        ("system", "Identify 5-10 key terms from the following topic, separated by comma
        ("user", "{topic}"),
    ])
    | llm
    | StrOutputParser()
)


# --- Build the Parallel + Synthesis Chain ---
# 1. Define the block of tasks to run in parallel. The results of these,
#    along with the original topic, will be fed into the next step.
```

```
map_chain = RunnableParallel(
    {
        "summary": summarize_chain,
        "questions": questions_chain,
        "key_terms": terms_chain,
        "topic": RunnablePassthrough(),  # Pass the original topic through
    }
)

# 2. Define the final synthesis prompt which will combine the parallel results.
synthesis_prompt = ChatPromptTemplate.from_messages([
    (
        "system",
        """Based on the following information:
        Summary: {summary}
        Related Questions: {questions}
        Key Terms: {key_terms}
        Synthesize a comprehensive answer."""
    ),
    ("user", "Original topic: {topic}"),
])

# 3. Construct the full chain by piping the parallel results directly
#    into the synthesis prompt, followed by the LLM and output parser.
full_parallel_chain = map_chain | synthesis_prompt | llm | StrOutputParser()


# --- Run the Chain ---
async def run_parallel_example(topic: str) -> None:
    """
    Asynchronously invokes the parallel processing chain with a specific topic
    and prints the synthesized result.

    Args:
```

```
        topic: The input topic to be processed by the LangChain chains.
    """
    if not llm:
        print("LLM not initialized. Cannot run example.")
        return

    print(f"\n--- Running Parallel LangChain Example for Topic: '{topic}' ---")
    try:
        # The input to `ainvoke` is the single 'topic' string,
        # then passed to each runnable in the `map_chain`.
        response = await full_parallel_chain.ainvoke(topic)
        print("\n--- Final Response ---")
        print(response)
    except Exception as e:
        print(f"\nAn error occurred during chain execution: {e}")


if __name__ == "__main__":
    test_topic = "The history of space exploration"
    # In Python 3.7+, asyncio.run is the standard way to run an async function.
    asyncio.run(run_parallel_example(test_topic))
```

The provided Python code implements a LangChain application designed for processing a given topic efficiently by leveraging parallel execution. Note that asyncio provides concurrency, not parallelism. It achieves this on a single thread by using an event loop that intelligently switches between tasks when one is idle (e.g., waiting for a network request). This creates the effect of multiple tasks progressing at once, but the code itself is still being executed by only one thread, constrained by Python's Global Interpreter Lock (GIL).

上述 Python 程式碼使用 LangChain 建構一個有效率處理主題的應用，透過平行執行提升效率。請注意 asyncio 提供的是併發而非真正平行，它在單一執行緒上運作，透過事件迴圈在任務等待（例如網路請求）時切換執行。這讓多個任務看似同時進行，但實際上仍在單一執行緒上受 Python 的 GIL 限制。

The code begins by importing essential modules from `langchain_openai` and

langchain_core, including components for language models, prompts, output parsing, and runnable structures. The code attempts to initialize a ChatOpenAI instance, specifically using the "gpt-4o-mini"model, with a specified temperature for controlling creativity. A try-except block is used for robustness during the language model initialization. Three independent LangChain "chains"are then defined, each designed to perform a distinct task on the input topic. The first chain is for summarizing the topic concisely, using a system message and a user message containing the topic placeholder. The second chain is configured to generate three interesting questions related to the topic. The third chain is set up to identify between 5 and 10 key terms from the input topic, requesting them to be comma-separated. Each of these independent chains consists of a ChatPromptTemplate tailored to its specific task, followed by the initialized language model and a StrOutputParser to format the output as a string.

程式先從 langchain_openai 與 langchain_core 匯入必要模組，包括語言模型、提示、輸出解析與 runnable 結構。接著嘗試初始化 ChatOpenAI（使用"gpt-4o-mini"模型並設定溫度），並用 try-except 確保初始化的穩健性。然後定義三個獨立的 LangChain「鏈」，各自針對輸入主題執行不同任務：第一個鏈負責精簡摘要；第二個鏈生成三個相關問題；第三個鏈識別 5 到 10 個關鍵詞並以逗號分隔。每個鏈都由針對任務的 ChatPromptTemplate、初始化的語言模型與 StrOutputParser 組成。

A RunnableParallel block is then constructed to bundle these three chains, allowing them to execute simultaneously. This parallel runnable also includes a RunnablePassthrough to ensure the original input topic is available for subsequent steps. A separate ChatPromptTemplate is defined for the final synthesis step, taking the summary, questions, key terms, and the original topic as input to generate a comprehensive answer. The full end-to-end processing chain, named full_parallel_chain, is created by sequencing the map_chain (the parallel block) into the synthesis prompt, followed by the language model and the output parser. An asynchronous function run_parallel_example is provided to demonstrate how to invoke this full_parallel_chain. This function takes the topic as input and uses invoke to run the asynchronous chain. Finally, the standard Python if __name__ == "__main__": block shows how to execute the run_parallel_example with a sample topic, in this case, "The history of space exploration", using asyncio.run to manage the asynchronous execution.

接著建立 RunnableParallel 區塊把三個鏈打包，使其同時執行。平行 runnable 也加入 RunnablePassthrough，以保留原始主題供後續步驟使用。接著為最終整合步驟定義 ChatPromptTemplate，輸入摘要、問題、關鍵詞與原始主題以產生完整答案。完整端到端流程 `full_parallel_chain` 由 `map_chain`（平行區塊）串接整合提示，再接語言模型與輸出解析。`run_parallel_example` 這個非同步函式示範如何呼叫 `full_parallel_chain`，它接收主題並以 ainvoke 執行非同步鏈。最後用標準的 `if __name__ == "__main__":` 區塊示範如何以 asyncio.run 執行，範例主題為「The history of space exploration」。

In essence, this code sets up a workflow where multiple LLM calls (for summarizing, questions, and terms) happen at the same time for a given topic, and their results are then combined by a final LLM call. This showcases the core idea of parallelization in an agentic workflow using LangChain.

總結而言，這段程式建立一個流程：針對同一主題，同時進行多個 LLM 呼叫（摘要、問題、關鍵詞），再由最後一個 LLM 呼叫整合結果。這正展現了 LangChain 中代理式流程的平行化核心概念。

Hands-On Code Example (Google ADK)

動手實作範例（Google ADK）

Okay, let's now turn our attention to a concrete example illustrating these concepts within the Google ADK framework. We'll examine how the ADK primitives, such as ParallelAgent and SequentialAgent, can be applied to build an agent flow that leverages concurrent execution for improved efficiency.

接著我們看一個 Google ADK 框架下的具體範例，說明這些概念如何落地。這裡會展示如何使用 ADK 的 Primitive（如 ParallelAgent、SequentialAgent）建立一個利用併發執行提升效率的代理流程。

```
from google.adk.agents import LlmAgent, ParallelAgent, SequentialAgent
from google.adk.tools import google_search

GEMINI_MODEL = "gemini-2.0-flash"

# --- 1. Define Researcher Sub-Agents (to run in parallel) ---
```

```
# Researcher 1: Renewable Energy
researcher_agent_1 = LlmAgent(
    name="RenewableEnergyResearcher",
    model=GEMINI_MODEL,
    instruction="""You are an AI Research Assistant specializing in energy. Research the
    description="Researches renewable energy sources.",
    tools=[google_search],
    # Store result in state for the merger agent
    output_key="renewable_energy_result",
)

# Researcher 2: Electric Vehicles
researcher_agent_2 = LlmAgent(
    name="EVResearcher",
    model=GEMINI_MODEL,
    instruction="""You are an AI Research Assistant specializing in transportation. Rese
    description="Researches electric vehicle technology.",
    tools=[google_search],
    # Store result in state for the merger agent
    output_key="ev_technology_result",
)

# Researcher 3: Carbon Capture
researcher_agent_3 = LlmAgent(
    name="CarbonCaptureResearcher",
    model=GEMINI_MODEL,
    instruction="""You are an AI Research Assistant specializing in climate solutions. R
    description="Researches carbon capture methods.",
    tools=[google_search],
    # Store result in state for the merger agent
    output_key="carbon_capture_result",
)
```

```
# --- 2. Create the ParallelAgent (Runs researchers concurrently) ---
# This agent orchestrates the concurrent execution of the researchers.
# It finishes once all researchers have completed and stored their results in state.
parallel_research_agent = ParallelAgent(
    name="ParallelWebResearchAgent",
    sub_agents=[researcher_agent_1, researcher_agent_2, researcher_agent_3],
    description="Runs multiple research agents in parallel to gather information.",
)


# --- 3. Define the Merger Agent (Runs after the parallel agents) ---
# This agent takes the results stored in the session state by the parallel agents
# and synthesizes them into a single, structured response with attributions.
merger_agent = LlmAgent(
    name="SynthesisAgent",
    model=GEMINI_MODEL,  # Or potentially a more powerful model if needed for synthesis
    instruction="""You are an AI Assistant responsible for combining research findings i

**Crucially:** Your entire response MUST be grounded *exclusively* on the information pr

**Input Summaries:**
*   **Renewable Energy:**
    {renewable_energy_result}
*   **Electric Vehicles:**
    {ev_technology_result}
*   **Carbon Capture:**
    {carbon_capture_result}

**Output Format:**
## Summary of Recent Sustainable Technology Advancements

### Renewable Energy Findings (Based on RenewableEnergyResearcher's findings)
[Synthesize and elaborate *only* on the renewable energy input summary provided above.]
```

```
### Electric Vehicle Findings (Based on EVResearcher's findings)
[Synthesize and elaborate *only* on the EV input summary provided above.]

### Carbon Capture Findings (Based on CarbonCaptureResearcher's findings)
[Synthesize and elaborate *only* on the carbon capture input summary provided above.]

### Overall Conclusion
[Provide a brief (1-2 sentence) concluding statement that connects *only* the findings p

Output *only* the structured report following this format. Do not include introductory c
""",
    description="Combines research findings from parallel agents into a structured, cite
    # No tools needed for merging
    # No output_key needed here, as its direct response is the final output of the seq
)


# --- 4. Create the SequentialAgent (Orchestrates the overall flow) ---
# This is the main agent that will be run. It first executes the ParallelAgent
# to populate the state, and then executes the MergerAgent to produce the final output
sequential_pipeline_agent = SequentialAgent(
    name="ResearchAndSynthesisPipeline",
    # Run parallel research first, then merge
    sub_agents=[parallel_research_agent, merger_agent],
    description="Coordinates parallel research and synthesizes the results.",
)

root_agent = sequential_pipeline_agent
```

This code defines a multi–agent system used to research and synthesize information on sustainable technology advancements. It sets up three LlmAgent instances to act as specialized researchers. `ResearcherAgent_1` focuses on renewable energy sources, `ResearcherAgent_2` researches electric vehicle technology, and `ResearcherAgent_3` investigates carbon capture methods. Each researcher agent is configured to use

a GEMINI_MODEL and the google_search tool. They are instructed to summarize their findings concisely (1–2 sentences) and store these summaries in the session state using output_key.

這段程式定義一個用於研究並整合永續科技進展的多代理人系統。它建立三個 LlmAgent 作為專門研究者：ResearcherAgent_1 關注再生能源、ResearcherAgent_2 研究電動車技術、ResearcherAgent_3 調查碳捕捉方法。每個研究代理人皆使用 GEMINI_MODEL 與 google_search 工具，並被指示用 1–2 句摘要研究結果，透過 output_key 存入 session state。

A ParallelAgent named ParallelWebResearchAgent is then created to run these three researcher agents concurrently. This allows the research to be conducted in parallel, potentially saving time. The ParallelAgent completes its execution once all its sub–agents (the researchers) have finished and populated the state.

接著建立名為 ParallelWebResearchAgent 的 ParallelAgent 以同時執行三個研究代理人。這讓研究可平行進行，節省時間。ParallelAgent 會在所有子代理人完成並寫入狀態後結束。

Next, a MergerAgent (also an LlmAgent) is defined to synthesize the research results. This agent takes the summaries stored in the session state by the parallel researchers as input. Its instruction emphasizes that the output must be strictly based only on the provided input summaries, prohibiting the addition of external knowledge. The Merg–erAgent is designed to structure the combined findings into a report with headings for each topic and a brief overall conclusion.

接著定義 MergerAgent（同樣是 LlmAgent）用來整合研究結果。此代理人以平行研究者寫入 session state 的摘要為輸入。指令強調輸出必須嚴格根據提供的摘要，不得加入外部知識。MergerAgent 會把整合結果整理成含各主題標題與簡短總結的報告。

Finally, a SequentialAgent named ResearchAndSynthesisPipeline is created to orches–trate the entire workflow. As the primary controller, this main agent first executes the ParallelAgent to perform the research. Once the ParallelAgent is complete, the SequentialAgent then executes the MergerAgent to synthesize the collected infor–mation. The sequential_pipeline_agent is set as the root_agent, representing the entry point for running this multi–agent system. The overall process is designed to efficiently gather information from multiple sources in parallel and then combine it into a single, structured report.

最後建立名為 ResearchAndSynthesisPipeline 的 SequentialAgent 來統整整體流程。這個主要代理人先執行 ParallelAgent 進行研究，待其完成後再執行 MergerAgent 進行整合。`sequential_pipeline_agent` 被設為 `root_agent`，代表系統的入口。整體流程設計為先平行蒐集多來源資訊，再整合成單一結構化報告。

At a Glance

一覽

What: Many agentic workflows involve multiple sub-tasks that must be completed to achieve a final goal. A purely sequential execution, where each task waits for the previous one to finish, is often inefficient and slow. This latency becomes a significant bottleneck when tasks depend on external I/O operations, such as calling different APIs or querying multiple databases. Without a mechanism for concurrent execution, the total processing time is the sum of all individual task durations, hindering the system's overall performance and responsiveness.

是什麼：許多代理式流程包含多個子任務，必須完成後才能達成最終目標。若完全採序列執行，後一任務需等待前一任務完成，常導致效率低下與延遲。當任務依賴外部 I/O（如呼叫多個 API 或查詢多個資料庫）時，延遲會成為顯著瓶頸。沒有併發機制時，總耗時等於各任務耗時相加，降低系統效能與回應性。

Why: The Parallelization pattern provides a standardized solution by enabling the simultaneous execution of independent tasks. It works by identifying components of a workflow, like tool usages or LLM calls, that do not rely on each other's immediate outputs. Agentic frameworks like LangChain and the Google ADK provide built-in constructs to define and manage these concurrent operations. For instance, a main process can invoke several sub-tasks that run in parallel and wait for all of them to complete before proceeding to the next step. By running these independent tasks at the same time rather than one after another, this pattern drastically reduces the total execution time.

為什麼：平行化模式透過同時執行獨立任務提供標準化解法。作法是識別流程中互不依賴的元件（如工具呼叫或 LLM 呼叫）。LangChain 與 Google ADK 等框架提供內建結構來定義與管理這些併發操作。例如主流程可同時啟動多個子任務，待全部完成後再進行下一步。此模式讓獨立任務同時執行而非依序執行，大幅縮短總時間。

Rule of thumb: Use this pattern when a workflow contains multiple independent operations that can run simultaneously, such as fetching data from several APIs, processing different chunks of data, or generating multiple pieces of content for later synthesis.

經驗法則：當流程包含多個可同時執行的獨立操作時使用此模式，例如從多個 API 擷取資料、處理不同資料區塊，或生成多段內容以供後續整合。

Visual summary:

視覺摘要：

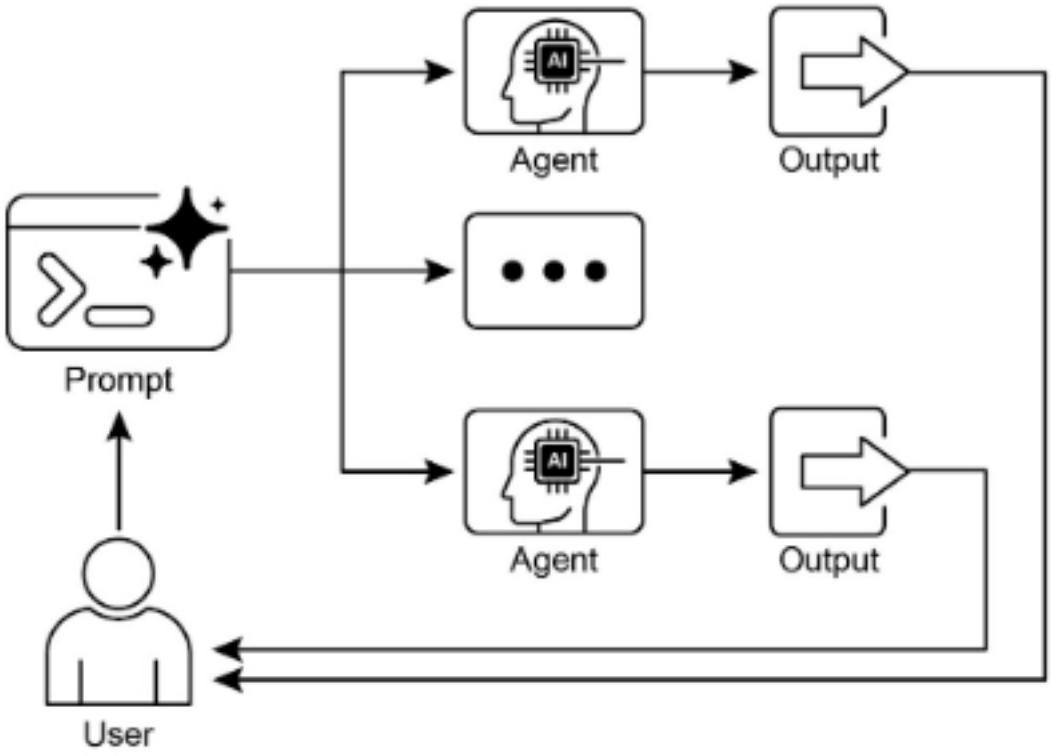

Figure 5: Parallelization Design Pattern

Fig.2: Parallelization design pattern

圖 2：平行化設計模式

Key Takeaways

重點整理

Here are the key takeaways:

重點如下：

- Parallelization is a pattern for executing independent tasks concurrently to improve efficiency.

- It is particularly useful when tasks involve waiting for external resources, such as API calls.

- The adoption of a concurrent or parallel architecture introduces substantial complexity and cost, impacting key development phases such as design, debugging, and system logging.

- Frameworks like LangChain and Google ADK provide built-in support for defining and managing parallel execution.

- In LangChain Expression Language (LCEL), RunnableParallel is a key construct for running multiple runnables side-by-side.

- Google ADK can facilitate parallel execution through LLM-Driven Delegation, where a Coordinator agent's LLM identifies independent sub-tasks and triggers their concurrent handling by specialized sub-agents.

- Parallelization helps reduce overall latency and makes agentic systems more responsive for complex tasks.
- 平行化是用於同時執行獨立任務以提升效率的模式。

- 當任務需等待外部資源（如 API 呼叫）時特別有用。

- 併發或平行架構會帶來顯著的複雜度與成本，影響設計、除錯與系統紀錄等階段。
- LangChain 與 Google ADK 提供內建支援，用於定義與管理平行執行。
- 在 LCEL 中，RunnableParallel 是並行多個 runnable 的關鍵構件。
- Google ADK 可透過 LLM 驅動的委派進行平行執行，由協調者 LLM 辨識獨立子任務並觸發專門子代理人同時處理。
- 平行化可降低整體延遲，讓代理式系統在複雜任務上更具回應性。

Conclusion

結論

The parallelization pattern is a method for optimizing computational workflows by concurrently executing independent sub-tasks. This approach reduces overall latency, particularly in complex operations that involve multiple model inferences or calls to external services.

平行化模式透過同時執行獨立子任務來優化計算流程，能降低整體延遲，特別適用於包含多次模型推論或外部服務呼叫的複雜操作。

Frameworks provide distinct mechanisms for implementing this pattern. In LangChain, constructs like RunnableParallel are used to explicitly define and execute multiple processing chains simultaneously. In contrast, frameworks like the Google Agent Developer Kit (ADK) can achieve parallelization through multi-agent delegation, where a primary coordinator model assigns different sub-tasks to specialized agents that can operate concurrently.

各框架提供不同機制來實作此模式。在 LangChain 中，RunnableParallel 用於明確定義並同時執行多條處理鏈。相較之下，Google ADK 可透過多代理人委派來達成平行化，由主要協調模型把不同子任務指派給可同時運作的專門代理人。

By integrating parallel processing with sequential (chaining) and conditional (routing) control flows, it becomes possible to construct sophisticated, high-performance computational systems capable of efficiently managing diverse and complex tasks.

將平行處理與序列（串接）及條件（路由）控制流程整合後，就能建構更精密、高效能的計算系統，有效管理多樣且複雜的任務。

References

參考資料

Here are some resources for further reading on the Parallelization pattern and related concepts:

以下為平行化模式與相關概念的延伸閱讀：

1. LangChain Expression Language (LCEL) Documentation (Parallelism): https://python.langchain.com/docs/concepts/lcel/
2. Google Agent Developer Kit (ADK) Documentation (Multi-Agent Systems): https://google.github.io/adk-docs/agents/multi-agents/

3. Python `asyncio` Documentation: https://docs.python.org/3/library/asyncio.html

---

Chapter 4: Reflection

第 4 章：反思

Reflection Pattern Overview

反思模式概覽

In the preceding chapters, we've explored fundamental agentic patterns: Chaining for sequential execution, Routing for dynamic path selection, and Parallelization for

concurrent task execution. These patterns enable agents to perform complex tasks more efficiently and flexibly. However, even with sophisticated workflows, an agent's initial output or plan might not be optimal, accurate, or complete. This is where the **Reflection** pattern comes into play.

在前幾章中，我們探討了代理式系統的基礎模式：用於序列執行的串接、用於動態路徑選擇的路由、以及用於並行任務執行的平行化。這些模式讓代理人能更有效率、更靈活地完成複雜任務。然而，即使有精密的流程，代理人的初始輸出或計畫仍可能不夠理想、不夠準確或不夠完整。這就是 **反思（Reflection）** 模式發揮作用的地方。

The Reflection pattern involves an agent evaluating its own work, output, or internal state and using that evaluation to improve its performance or refine its response. It's a form of self-correction or self-improvement, allowing the agent to iteratively refine its output or adjust its approach based on feedback, internal critique, or comparison against desired criteria. Reflection can occasionally be facilitated by a separate agent whose specific role is to analyze the output of an initial agent.

反思模式是指代理人評估自身的工作、輸出或內部狀態，並用這些評估來改進表現或修正回應。這是一種自我修正或自我提升的形式，讓代理人能根據回饋、內部批判或與目標標準的比較，反覆精煉輸出或調整方法。有時反思也可由另一個專職代理人來進行，專門負責分析初始代理人的輸出。

Unlike a simple sequential chain where output is passed directly to the next step, or routing which chooses a path, reflection introduces a feedback loop. The agent doesn't just produce an output; it then examines that output (or the process that generated it), identifies potential issues or areas for improvement, and uses those insights to generate a better version or modify its future actions.

與單純將輸出傳到下一步的序列鏈不同，也不同於路由的路徑選擇，反思引入了一個回饋迴圈。代理人不只是產出結果，而是會檢視該輸出（或產生輸出的流程），找出可能問題或可改進之處，再用這些洞見生成更好的版本或調整後續行動。

The process typically involves:

此流程通常包含：

1. **Execution:** The agent performs a task or generates an initial output.

2. **Evaluation/Critique:** The agent (often using another LLM call or a set of rules)

analyzes the result from the previous step. This evaluation might check for factual accuracy, coherence, style, completeness, adherence to instructions, or other relevant criteria.

3. **Reflection/Refinement:** Based on the critique, the agent determines how to improve. This might involve generating a refined output, adjusting parameters for a subsequent step, or even modifying the overall plan.

4. **Iteration (Optional but common):** The refined output or adjusted approach can then be executed, and the reflection process can repeat until a satisfactory result is achieved or a stopping condition is met.

5. **執行：**代理人執行任務或產生初始輸出。

6. **評估/批判：**代理人（常透過另一個 LLM 呼叫或規則集）分析前一步結果，檢查事實正確性、連貫性、風格、完整度、是否符合指令或其他相關標準。

7. **反思/精煉：**根據批判結果決定如何改進，例如產生精煉輸出、調整下一步參數，甚至修改整體計畫。

8. **迭代（可選但常見）：**重新執行精煉後的輸出或調整後的方法，重複反思流程直到滿足品質或達到停止條件。

A key and highly effective implementation of the Reflection pattern separates the process into two distinct logical roles: a Producer and a Critic. This is often called the "Generator–Critic"or "Producer–Reviewer"model. While a single agent can perform self–reflection, using two specialized agents (or two separate LLM calls with distinct system prompts) often yields more robust and unbiased results.

一個關鍵且高效的實作方式，是把反思流程分成兩個明確角色：產出者（Producer）與評論者（Critic）。這通常稱為「生成者–評論者」或「產出者–審閱者」模型。雖然單一代理人也能自我反思，但使用兩個專門代理人（或兩次不同 system prompt 的 LLM 呼叫）通常能得到更穩健且更客觀的結果。

1. The Producer Agent: This agent's primary responsibility is to perform the initial

execution of the task. It focuses entirely on generating the content, whether it's writing code, drafting a blog post, or creating a plan. It takes the initial prompt and produces the first version of the output.

2. The Critic Agent: This agent's sole purpose is to evaluate the output generated by the Producer. It is given a different set of instructions, often a distinct persona (e.g., "You are a senior software engineer," "You are a meticulous fact-checker"). The Critic's instructions guide it to analyze the Producer's work against specific criteria, such as factual accuracy, code quality, stylistic requirements, or completeness. It is designed to find flaws, suggest improvements, and provide structured feedback.

3. 產出者代理人：此代理人的主要責任是完成任務的初始執行，專注於產出內容，無論是寫程式碼、撰寫部落格文章或擬定計畫。它接收初始提示並產出第一版結果。

4. 評論者代理人：此代理人唯一目的是評估產出者的輸出。它會被賦予不同指令，常以不同角色設定（如「你是資深軟體工程師」或「你是嚴謹的事實查核者」）。評論者依特定標準分析產出者的作品，例如事實正確性、程式碼品質、風格要求或完整度，並找出缺陷、提出改進建議與結構化回饋。

This separation of concerns is powerful because it prevents the "cognitive bias"of an agent reviewing its own work. The Critic agent approaches the output with a fresh perspective, dedicated entirely to finding errors and areas for improvement. The feedback from the Critic is then passed back to the Producer agent, which uses it as a guide to generate a new, refined version of the output. The provided LangChain and ADK code examples both implement this two-agent model: the LangChain example uses a specific `reflector_prompt` to create a critic persona, while the ADK example explicitly defines a producer and a reviewer agent.

這種角色分離很有力，因為它能避免代理人自我審視時產生「認知偏誤」。評論者以全新視角檢視輸出，專注於找錯與改進點。評論回饋會傳回產出者，作為生成新版輸出的指引。文中的 LangChain 與 ADK 範例都實作了此雙代理人模型：LangChain 以 `reflector_prompt` 建立評論者角色，ADK 則明確定義產出者與審閱者代理人。

Implementing reflection often requires structuring the agent's workflow to include these feedback loops. This can be achieved through iterative loops in code, or using frameworks that support state management and conditional transitions based on evaluation results. While a single step of evaluation and refinement can be imple-

mented within either a LangChain/LangGraph, or ADK, or Crew.AI chain, true iterative reflection typically involves more complex orchestration.

反思的實作通常需要在工作流程中加入回饋迴圈。可透過程式中的迭代迴圈實現，或使用支援狀態管理與基於評估結果的條件轉移之框架。雖然單次評估與修正可在 LangChain/LangGraph、ADK 或 Crew.AI 鏈中實作，但真正的迭代式反思往往需要更複雜的編排。

The Reflection pattern is crucial for building agents that can produce high–quality outputs, handle nuanced tasks, and exhibit a degree of self–awareness and adaptability. It moves agents beyond simply executing instructions towards a more sophisticated form of problem–solving and content generation.

反思模式是打造高品質輸出、處理細緻任務並具備一定自我覺察與適應力的代理人的關鍵。它讓代理人超越單純執行指令，邁向更成熟的問題解決與內容生成能力。

The intersection of reflection with goal setting and monitoring (see Chapter 11) is worth noticing. A goal provides the ultimate benchmark for the agent's self–evaluation, while monitoring tracks its progress. In a number of practical cases, Reflection then might act as the corrective engine, using monitored feedback to analyze deviations and adjust its strategy. This synergy transforms the agent from a passive executor into a purposeful system that adaptively works to achieve its objectives.

值得注意的是反思與目標設定與監控（見第 11 章）的交集。目標提供自我評估的最終標準，監控用以追蹤進度。在許多實務案例中，反思可作為修正引擎，透過監控回饋分析偏差並調整策略。這種協同讓代理人從被動執行者轉變為能自適應地朝目標努力的系統。

Furthermore, the effectiveness of the Reflection pattern is significantly enhanced when the LLM keeps a memory of the conversation (see Chapter 8). This conversational history provides crucial context for the evaluation phase, allowing the agent to assess its output not just in isolation, but against the backdrop of previous interactions, user feedback, and evolving goals. It enables the agent to learn from past critiques and avoid repeating errors. Without memory, each reflection is a self–contained event; with memory, reflection becomes a cumulative process where each cycle builds upon the last, leading to more intelligent and context–aware refinement.

此外，當 LLM 能保留對話記憶（見第 8 章）時，反思的效果會顯著提升。對話歷史為評估階段提供關鍵脈絡，使代理人能在先前互動、使用者回饋與目標演變的背景下檢視輸

出，而非僅就當前輸出孤立判斷。這讓代理人能從過往批評中學習，避免重複錯誤。沒有記憶時，每次反思都是獨立事件；有記憶時，反思成為累積性過程，每次循環都建立在前一次之上，帶來更聰明、更具情境感知的精煉。

Practical Applications & Use Cases

實務應用與使用情境

The Reflection pattern is valuable in scenarios where output quality, accuracy, or adherence to complex constraints is critical:

反思模式適用於對輸出品質、正確性或複雜約束遵循度要求嚴格的情境：

1. Creative Writing and Content Generation

1. 創意寫作與內容生成

Refining generated text, stories, poems, or marketing copy.

用於精煉生成的文本、故事、詩或行銷文案。

- **Use Case:** An agent writing a blog post.
  - **Reflection:** Generate a draft, critique it for flow, tone, and clarity, then rewrite based on the critique. Repeat until the post meets quality standards.
  - **Benefit:** Produces more polished and effective content.
- **使用情境：**代理人撰寫部落格文章。
  - **反思：**先產生草稿，再就流程、語氣與清晰度批判，依批判修寫，直到達到品質標準。
  - **效益：**產出更精緻、有效的內容。

2. Code Generation and Debugging

2. 程式碼生成與除錯

Writing code, identifying errors, and fixing them.

撰寫程式、找錯並修正。

- **Use Case:** An agent writing a Python function.
  - **Reflection:** Write initial code, run tests or static analysis, identify errors or inefficiencies, then modify the code based on the findings.
  - **Benefit:** Generates more robust and functional code.
- **使用情境：**代理人撰寫 Python 函式。
  - **反思：**先寫初版，再執行測試或靜態分析，找出錯誤或低效之處並修正。
  - **效益：**產生更穩健、可用的程式碼。

3. Complex Problem Solving

3. 複雜問題解決

Evaluating intermediate steps or proposed solutions in multi-step reasoning tasks.

在多步推理任務中評估中間步驟或提出的解法。

- **Use Case:** An agent solving a logic puzzle.
  - **Reflection:** Propose a step, evaluate if it leads closer to the solution or introduces contradictions, backtrack or choose a different step if needed.
  - **Benefit:** Improves the agent's ability to navigate complex problem spaces.
- **使用情境：**代理人解邏輯謎題。
  - **反思：**提出一步，評估是否接近解答或引入矛盾，需要時回溯或改走其他步驟。
  - **效益：**提升代理人探索複雜問題空間的能力。

4. Summarization and Information Synthesis

4. 摘要與資訊整合

Refining summaries for accuracy, completeness, and conciseness.

精煉摘要以提升正確性、完整度與精簡度。

- **Use Case:** An agent summarizing a long document.
  - **Reflection:** Generate an initial summary, compare it against key points in the original document, refine the summary to include missing information or improve accuracy.
  - **Benefit:** Creates more accurate and comprehensive summaries.
- **使用情境：**代理人摘要長文件。
  - **反思：**產生初版摘要，與原文重點比對，補上遺漏或修正不準之處。
  - **效益：**產生更準確、更完整的摘要。

5. Planning and Strategy

5. 規劃與策略

Evaluating a proposed plan and identifying potential flaws or improvements.

評估提案並找出缺陷或改進點。

- **Use Case:** An agent planning a series of actions to achieve a goal.
  - **Reflection:** Generate a plan, simulate its execution or evaluate its feasibility against constraints, revise the plan based on the evaluation.
  - **Benefit:** Develops more effective and realistic plans.
- **使用情境：**代理人規劃達成目標的行動序列。
  - **反思：**產生計畫，模擬執行或以限制條件檢核可行性，依評估結果修訂計畫。
  - **效益：**制定更有效、更務實的計畫。

6. Conversational Agents

6. 對話式代理人

Reviewing previous turns in a conversation to maintain context, correct misunderstandings, or improve response quality.

回顧對話歷史以維持脈絡、修正誤解或提升回應品質。

- **Use Case:** A customer support chatbot.
  - **Reflection:** After a user response, review the conversation history and the last generated message to ensure coherence and address the user's latest input accurately.
  - **Benefit:** Leads to more natural and effective conversations.
- **使用情境：**客服聊天機器人。
  - **反思：**使用者回覆後，回顧對話歷史與最後訊息，確保一致性並準確回應最新輸入。
  - **效益：**讓對話更自然且更有效。

Reflection adds a layer of meta-cognition to agentic systems, enabling them to learn from their own outputs and processes, leading to more intelligent, reliable, and high-quality results.

反思為代理式系統加入一層「後設認知」，使其能從自身輸出與流程中學習，進而產生更聰明、更可靠且高品質的結果。

Hands-On Code Example (LangChain)

動手實作範例（LangChain）

The implementation of a complete, iterative reflection process necessitates mechanisms for state management and cyclical execution. While these are handled natively in graph-based frameworks like LangGraph or through custom procedural code, the fundamental principle of a single reflection cycle can be demonstrated effectively using the compositional syntax of LCEL (LangChain Expression Language).

完整的迭代式反思流程需要狀態管理與循環執行機制。這可由 LangGraph 等圖形框架原生處理，或用自訂流程程式實作；但單次反思循環的基本原則，也能用 LCEL（LangChain Expression Language）的組合語法清楚示範。

This example implements a reflection loop using the Langchain library and OpenAI's GPT-4o model to iteratively generate and refine a Python function that calculates the factorial of a number. The process starts with a task prompt, generates initial code, and then repeatedly reflects on the code based on critiques from a simulated

senior software engineer role, refining the code in each iteration until the critique stage determines the code is perfect or a maximum number of iterations is reached. Finally, it prints the resulting refined code.

此範例使用 Langchain 與 OpenAI 的 GPT-4o 模型實作反思迴圈，反覆生成與精煉計算階乘的 Python 函式。流程從任務提示開始，產生初版程式碼，再由模擬的資深工程師角色提出批判並反覆修正，直到批判階段判定程式碼完美或達到最大迭代次數。最後輸出精煉後的程式碼。

First, ensure you have the necessary libraries installed:

首先，請確認已安裝必要套件：

```
pip install langchain langchain-community langchain-openai
```

You will also need to set up your environment with your API key for the language model you choose (e.g., OpenAI, Google Gemini, Anthropic).

你也需要設定所選模型供應商（如 OpenAI、Google Gemini 或 Anthropic）的 API 金鑰。

```
import os
from dotenv import load_dotenv
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.messages import SystemMessage, HumanMessage


# --- Configuration ---
# Load environment variables from .env file (for OPENAI_API_KEY)
load_dotenv()

# Check if the API key is set
if not os.getenv("OPENAI_API_KEY"):
    raise ValueError("OPENAI_API_KEY not found in .env file. Please add it.")

# Initialize the Chat LLM. We use gpt-4o for better reasoning.
# A lower temperature is used for more deterministic outputs.
llm = ChatOpenAI(model="gpt-4o", temperature=0.1)
```

```python
def run_reflection_loop():
    """
    Demonstrates a multi-step AI reflection loop to progressively improve a Python fun
    """
    # --- The Core Task ---
    task_prompt = """
    Your task is to create a Python function named `calculate_factorial`.
    This function should do the following:
    1.  Accept a single integer `n` as input.
    2.  Calculate its factorial (n!).
    3.  Include a clear docstring explaining what the function does.
    4.  Handle edge cases: The factorial of 0 is 1.
    5.  Handle invalid input: Raise a ValueError if the input is a negative number.
    """

    # --- The Reflection Loop ---
    max_iterations = 3
    current_code = ""

    # We will build a conversation history to provide context in each step.
    message_history = [HumanMessage(content=task_prompt)]

    for i in range(max_iterations):
        print("\n" + "=" * 25 + f" REFLECTION LOOP: ITERATION {i + 1} " + "=" * 25)

        # --- 1. GENERATE / REFINE STAGE ---
        # In the first iteration, it generates. In subsequent iterations, it refines.
        if i == 0:
            print("\n>>> STAGE 1: GENERATING initial code...")
            # The first message is just the task prompt.
            response = llm.invoke(message_history)
            current_code = response.content
        else:
```

```
        print("\n>>> STAGE 1: REFINING code based on previous critique...")
        # The message history now contains the task,
        # the last code, and the last critique.
        # We instruct the model to apply the critiques.
        message_history.append(HumanMessage(content="Please refine the code using th
        response = llm.invoke(message_history)
        current_code = response.content

    print("\n--- Generated Code (v" + str(i + 1) + ") ---\n" + current_code)
    message_history.append(response)  # Add the generated code to history

    # --- 2. REFLECT STAGE ---
    print("\n>>> STAGE 2: REFLECTING on the generated code...")
    # Create a specific prompt for the reflector agent.
    # This asks the model to act as a senior code reviewer.
    reflector_prompt = [
        SystemMessage(content="""
            You are a senior software engineer and an expert
            in Python.
            Your role is to perform a meticulous code review.
            Critically evaluate the provided Python code based
            on the original task requirements.
            Look for bugs, style issues, missing edge cases,
            and areas for improvement.
            If the code is perfect and meets all requirements,
            respond with the single phrase 'CODE_IS_PERFECT'.
            Otherwise, provide a bulleted list of your critiques.
        """),
        HumanMessage(content=f"Original Task:\n{task_prompt}\n\nCode to Review:\n{cu
    ]

    critique_response = llm.invoke(reflector_prompt)
    critique = critique_response.content
```

```
        # --- 3. STOPPING CONDITION ---
        if "CODE_IS_PERFECT" in critique:
            print("\n--- Critique ---\nNo further critiques found. The code is satisfact
            break

        print("\n--- Critique ---\n" + critique)
        # Add the critique to the history for the next refinement loop.
        message_history.append(HumanMessage(content=f"Critique of the previous code:\n{c

    print("\n" + "=" * 30 + " FINAL RESULT " + "=" * 30)
    print("\nFinal refined code after the reflection process:\n")
    print(current_code)


if __name__ == "__main__":
    run_reflection_loop()
```

The code begins by setting up the environment, loading API keys, and initializing a powerful language model like GPT–4o with a low temperature for focused outputs. The core task is defined by a prompt asking for a Python function to calculate the factorial of a number, including specific requirements for docstrings, edge cases (factorial of 0), and error handling for negative input. The `run_reflection_loop` function orchestrates the iterative refinement process. Within the loop, in the first iteration, the language model generates initial code based on the task prompt. In subsequent iterations, it refines the code based on critiques from the previous step. A separate "reflector"role, also played by the language model but with a different system prompt, acts as a senior software engineer to critique the generated code against the original task requirements. This critique is provided as a bulleted list of issues or the phrase `CODE_IS_PERFECT` if no issues are found. The loop continues until the critique indicates the code is perfect or a maximum number of iterations is reached. The conversation history is maintained and passed to the language model in each step to provide context for both generation/refinement and reflection stages. Finally, the script prints the last generated code version after the loop concludes.

此程式碼先設定環境、載入 API 金鑰，並以較低溫度初始化 GPT–4o 以取得更穩定的輸出。核心任務以提示定義：撰寫計算階乘的 Python 函式，需包含 docstring、處理 0 的

邊界情況、以及對負數輸入丟出錯誤。run_reflection_loop 會協調迭代式精煉流程：第一輪由模型依任務提示產生初版程式碼；後續迭代則依上一輪的評論進行修正。另一個「評論者」角色（同樣由 LLM 扮演，但用不同 system prompt）以資深工程師視角批判程式碼是否符合需求。評論以條列問題或 CODE_IS_PERFECT 呈現。迴圈持續到程式碼被判定完美或達最大迭代次數為止。每輪都維持對話歷史，以提供生成/精煉與反思階段所需脈絡。最後輸出最終精煉版本。

Hands-On Code Example (ADK)

動手實作範例（ADK）

Let's now look at a conceptual code example implemented using the Google ADK. Specifically, the code showcases this by employing a Generator-Critic structure, where one component (the Generator) produces an initial result or plan, and another component (the Critic) provides critical feedback or a critique, guiding the Generator towards a more refined or accurate final output.

接著看一個使用 Google ADK 的概念性範例，透過生成者–評論者（Generator–Critic）結構實作反思：生成者先產出初稿或計畫，評論者提供批判回饋，引導生成者產生更精煉或更準確的結果。

```
from google.adk.agents import SequentialAgent, LlmAgent


# The first agent generates the initial draft.
generator = LlmAgent(
    name="DraftWriter",
    description="Generates initial draft content on a given subject.",
    instruction="Write a short, informative paragraph about the user's subject.",
    output_key="draft_text",  # The output is saved to this state key.
)

# The second agent critiques the draft from the first agent.
reviewer = LlmAgent(
    name="FactChecker",
    description="Reviews a given text for factual accuracy and provides a structured cri
```

```
    instruction="""
    You are a meticulous fact-checker.
    1. Read the text provided in the state key 'draft_text'.
    2. Carefully verify the factual accuracy of all claims.
    3. Your final output must be a dictionary containing two keys:
       - "status": A string, either "ACCURATE" or "INACCURATE".
       - "reasoning": A string providing a clear explanation for your status, citing spe
    """,
    output_key="review_output",  # The structured dictionary is saved here.
)

# The SequentialAgent ensures the generator runs before the reviewer.
review_pipeline = SequentialAgent(
    name="WriteAndReview_Pipeline",
    sub_agents=[generator, reviewer],
)

# Execution Flow:
# 1. generator runs -> saves its paragraph to state['draft_text'].
# 2. reviewer runs -> reads state['draft_text'] and saves its dictionary output to sta
```

This code demonstrates the use of a sequential agent pipeline in Google ADK for generating and reviewing text. It defines two LlmAgent instances: generator and reviewer. The generator agent is designed to create an initial draft paragraph on a given subject. It is instructed to write a short and informative piece and saves its output to the state key `draft_text`. The reviewer agent acts as a fact-checker for the text produced by the generator. It is instructed to read the text from `draft_text` and verify its factual accuracy. The reviewer's output is a structured dictionary with two keys: status and reasoning. status indicates if the text is "ACCURATE"or "IN-ACCURATE", while reasoning provides an explanation for the status. This dictionary is saved to the state key `review_output`. A SequentialAgent named `review_pipeline` is created to manage the execution order of the two agents. It ensures that the generator runs first, followed by the reviewer. The overall execution flow is that the generator produces text, which is then saved to the state. Subsequently, the reviewer reads this text from the state, performs its fact-checking, and saves its findings (the

status and reasoning) back to the state. This pipeline allows for a structured process of content creation and review using separate agents.

Note: An alternative implementation utilizing ADK's LoopAgent is also available for those interested.

這段程式碼示範 Google ADK 的序列代理管線，用於生成與審核文本。它定義兩個 LlmAgent：generator 與 reviewer。generator 產出某主題的初稿段落，並存到 `draft_text`。reviewer 作為事實查核者，讀取 `draft_text` 並檢查內容正確性，輸出包含 `status` 與 `reasoning` 的結構化字典，存到 `review_output`。名為 `review_pipeline` 的 SequentialAgent 控制執行順序，確保 generator 先執行、reviewer 後執行。整體流程為：生成文字 → 存入狀態 → 審核文字 → 回存審核結果。這讓內容生成與審查能以分工代理人進行結構化流程。

註：也可使用 ADK 的 LoopAgent 實作替代方案。

Before concluding, it's important to consider that while the Reflection pattern significantly enhances output quality, it comes with important trade-offs. The iterative process, though powerful, can lead to higher costs and latency, since every refinement loop may require a new LLM call, making it suboptimal for time-sensitive applications. Furthermore, the pattern is memory-intensive; with each iteration, the conversational history expands, including the initial output, critique, and subsequent refinements.

在結尾前需留意：反思模式雖能大幅提升輸出品質，但也有重要取捨。迭代過程雖強大，卻可能造成更高成本與延遲，因每次精煉迴圈都需新的 LLM 呼叫，對時效敏感的應用不理想。此外，反思也耗記憶體；每次迭代都會擴張對話歷史，包含初始輸出、批判與後續修正。

At Glance

一覽

What: An agent's initial output is often suboptimal, suffering from inaccuracies, incompleteness, or a failure to meet complex requirements. Basic agentic workflows lack a built-in process for the agent to recognize and fix its own errors. This is solved by having the agent evaluate its own work or, more robustly, by introducing

a separate logical agent to act as a critic, preventing the initial response from being the final one regardless of quality.

是什麼：代理人的初始輸出常不理想，可能不準確、不完整或未符合複雜需求。基本的代理流程缺乏讓代理人辨識並修正自身錯誤的內建機制。此問題可透過讓代理人自我評估，或更穩健地加入獨立評論者代理人來解決，避免初始回應因品質不足而直接成為最終結果。

Why: The Reflection pattern offers a solution by introducing a mechanism for self–correction and refinement. It establishes a feedback loop where a “producer”agent generates an output, and then a “critic”agent (or the producer itself) evaluates it against predefined criteria. This critique is then used to generate an improved version. This iterative process of generation, evaluation, and refinement progressively enhances the quality of the final result, leading to more accurate, coherent, and reliable outcomes.

為什麼：反思模式透過引入自我修正與精煉機制提供解法。它建立一個回饋迴圈：由「產出者」生成結果，再由「評論者」（或產出者自己）依預先定義標準評估。此批判再用於產生改良版本。透過生成—評估—精煉的迭代流程，最終成果品質逐步提升，帶來更準確、連貫且可靠的結果。

Rule of thumb: Use the Reflection pattern when the quality, accuracy, and detail of the final output are more important than speed and cost. It is particularly effective for tasks like generating polished long–form content, writing and debugging code, and creating detailed plans. Employ a separate critic agent when tasks require high objectivity or specialized evaluation that a generalist producer agent might miss.

經驗法則：當最終輸出的品質、正確性與細節比速度與成本更重要時使用反思模式。它特別適合生成長篇內容、寫作與除錯程式碼、或建立詳細計畫。當任務需要高度客觀性或專業評估時，應使用獨立評論者代理人，以補足通用型產出者可能忽略之處。

Visual summary:

視覺摘要：

Fig. 1: Reflection design pattern, self–reflection

圖 1：反思設計模式—自我反思

Fig.2: Reflection design pattern, producer and critique agent

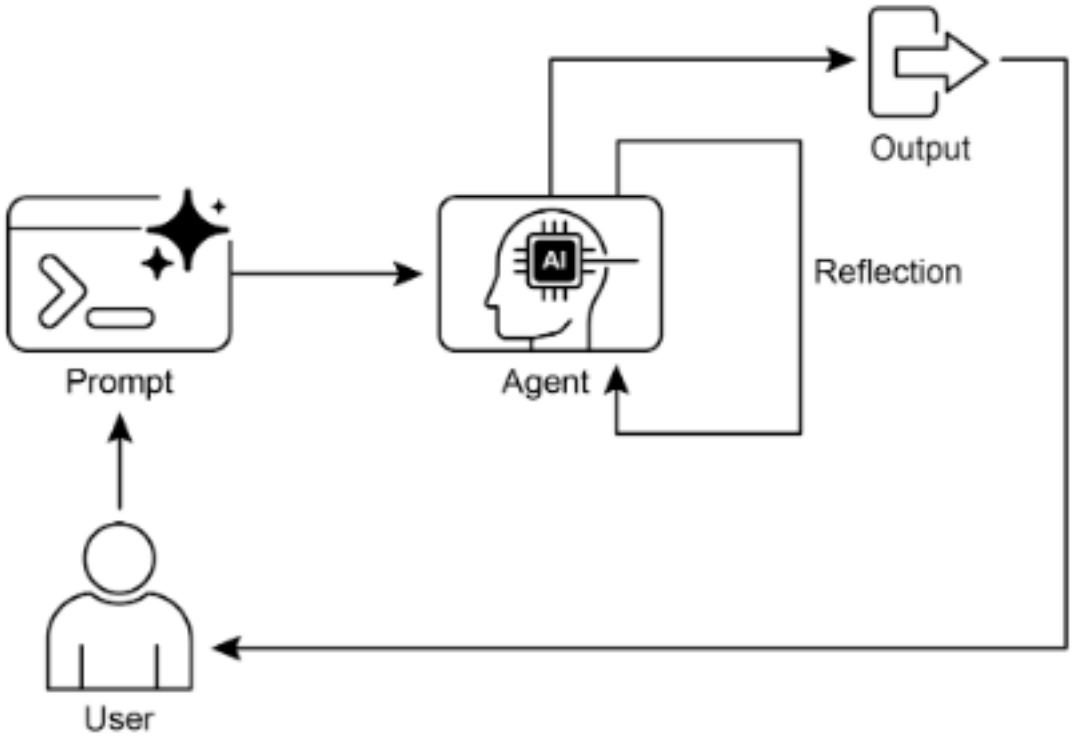

Figure 6: Reflection Design Pattern, Self–Reflection

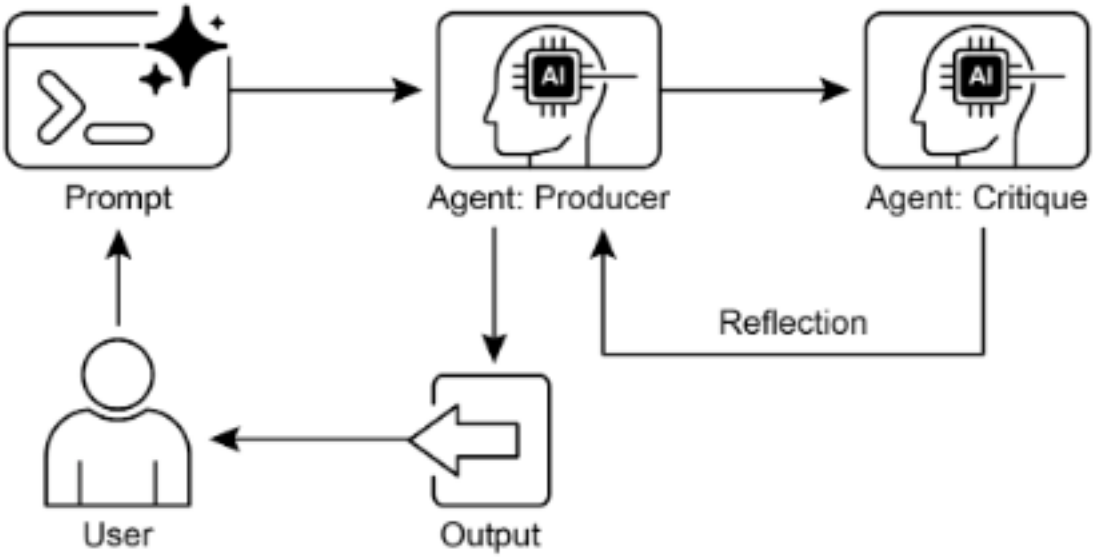

Figure 7: Reflection Design Pattern, Producer and Critique Agent

圖 2：反思設計模式—產出者與評論者

Key Takeaways

重點整理

- The primary advantage of the Reflection pattern is its ability to iteratively self–correct and refine outputs, leading to significantly higher quality, accuracy, and adherence to complex instructions.

- It involves a feedback loop of execution, evaluation/critique, and refinement. Reflection is essential for tasks requiring high–quality, accurate, or nuanced outputs.

- A powerful implementation is the Producer–Critic model, where a separate agent (or prompted role) evaluates the initial output. This separation of concerns enhances objectivity and allows for more specialized, structured feedback.

- However, these benefits come at the cost of increased latency and compu–tational expense, along with a higher risk of exceeding the model's context window or being throttled by API services.

- While full iterative reflection often requires stateful workflows (like LangGraph), a single reflection step can be implemented in LangChain using LCEL to pass output for critique and subsequent refinement.

- Google ADK can facilitate reflection through sequential workflows where one agent's output is critiqued by another agent, allowing for subsequent refinement steps.

- This pattern enables agents to perform self–correction and enhance their per–

formance over time.

- 反思模式的主要優勢是能迭代自我修正與精煉輸出，使品質、正確性與對複雜指令的遵循大幅提升。

- 它包含執行、評估/批判與精煉的回饋迴圈，對需要高品質、準確或細緻輸出的任務特別重要。

- 一個強力實作是產出者–評論者模型，由獨立代理人（或指定角色）評估初始輸出；角色分離提升客觀性並帶來更專門、結構化的回饋。

- 這些優點伴隨成本：延遲與計算費用增加，且更容易超過模型 context window 或遭 API 服務節流。

- 完整迭代反思通常需要具狀態流程（如 LangGraph），但單次反思可用 LangChain 的 LCEL 來傳遞輸出並做批判與修正。

- Google ADK 可透過序列流程實現反思：一個代理人的輸出由另一個代理人批判，並可接續精煉。

- 此模式讓代理人能自我修正並隨時間提升表現。

Conclusion

結論

The reflection pattern provides a crucial mechanism for self–correction within an agent's workflow, enabling iterative improvement beyond a single–pass execution. This is achieved by creating a loop where the system generates an output, evaluates it against specific criteria, and then uses that evaluation to produce a refined result. This evaluation can be performed by the agent itself (self–reflection) or, often more effectively, by a distinct critic agent, which represents a key architectural choice within the pattern.

反思模式為代理人的流程提供關鍵的自我修正機制，使其能在單次產出之外進行迭代式改進。其做法是建立一個迴圈：系統產生輸出、依特定標準評估，並根據評估產生精煉結果。評估可由代理人自我反思完成，但更常見且有效的做法是由獨立評論者代理人執行，這也是此模式的重要架構選擇之一。

While a fully autonomous, multi-step reflection process requires a robust architecture for state management, its core principle is effectively demonstrated in a single generate-critique-refine cycle. As a control structure, reflection can be integrated with other foundational patterns to construct more robust and functionally complex agentic systems.

雖然完整的自主多步反思流程需要穩健的狀態管理架構，但其核心原則在單次「生成—批判—精煉」循環中就能有效呈現。作為控制結構，反思可與其他基礎模式整合，以建構更穩健、功能更複雜的代理式系統。

References

參考資料

Here are some resources for further reading on the Reflection pattern and related concepts:

以下為反思模式與相關概念的延伸閱讀：

1. Training Language Models to Self-Correct via Reinforcement Learning, https://arxiv.org/abs/2409.12917
2. LangChain Expression Language (LCEL) Documentation: https://python.langchain.com/docs/introduction/
3. LangGraph Documentation:https://www.langchain.com/langgraph
4. Google Agent Developer Kit (ADK) Documentation (Multi-Agent Systems): https://google.github.io/adk-docs/agents/multi-agents/

Chapter 5: Tool Use (Function Calling)

第 5 章：工具使用（函式呼叫）

Tool Use Pattern Overview

工具使用模式概覽

So far, we've discussed agentic patterns that primarily involve orchestrating interactions between language models and managing the flow of information within the agent's internal workflow (Chaining, Routing, Parallelization, Reflection). However, for agents to be truly useful and interact with the real world or external systems, they need the ability to use Tools.

到目前為止，我們討論的代理式模式主要著重於協調語言模型之間的互動，並管理代理人內部流程中的資訊流（串接、路由、平行化、反思）。然而，若要讓代理人真正有用、能與真實世界或外部系統互動，就必須具備使用工具的能力。

The Tool Use pattern, often implemented through a mechanism called Function Calling, enables an agent to interact with external APIs, databases, services, or even execute code. It allows the LLM at the core of the agent to decide when and how to use a specific external function based on the user's request or the current state of the task.

工具使用模式通常透過「函式呼叫（Function Calling）」機制實作，使代理人能與外部 API、資料庫、服務互動，甚至執行程式碼。它讓代理人核心的 LLM 能依據使用者需求或當前任務狀態，判斷何時、如何使用特定外部函式。

The process typically involves:

流程通常包含：

1. **Tool Definition:** External functions or capabilities are defined and described to the LLM. This description includes the function's purpose, its name, and the parameters it accepts, along with their types and descriptions.

2. **LLM Decision:** The LLM receives the user's request and the available tool definitions. Based on its understanding of the request and the tools, the LLM

decides if calling one or more tools is necessary to fulfill the request.

3. **Function Call Generation:** If the LLM decides to use a tool, it generates a structured output (often a JSON object) that specifies the name of the tool to call and the arguments (parameters) to pass to it, extracted from the user's request.

4. **Tool Execution:** The agentic framework or orchestration layer intercepts this structured output. It identifies the requested tool and executes the actual external function with the provided arguments.

5. **Observation/Result:** The output or result from the tool execution is returned to the agent.

6. **LLM Processing (Optional but common):** The LLM receives the tool's output as context and uses it to formulate a final response to the user or decide on the next step in the workflow (which might involve calling another tool, reflecting, or providing a final answer).
7. **工具定義：**將外部函式或能力定義並描述給 LLM，包括用途、名稱、參數及其型別與說明。

8. **LLM 決策：** LLM 接收使用者需求與工具定義，根據理解判斷是否需要呼叫一個或多個工具。

9. **函式呼叫生成：**若決定使用工具，LLM 會生成結構化輸出（常為 JSON），指定要呼叫的工具名稱與參數。

10. **工具執行：**代理框架或編排層攔截此結構化輸出，識別工具並以參數執行外部函式。

11. **觀察/結果：**工具執行的輸出或結果回傳給代理人。

12. **LLM 後處理（可選但常見）：** LLM 以工具輸出作為脈絡，形成最終回應或決定下一步（例如再呼叫工具、進行反思或給出最終答案）。

This pattern is fundamental because it breaks the limitations of the LLM's training data and allows it to access up-to-date information, perform calculations it can't do internally, interact with user-specific data, or trigger real-world actions. Function calling is the technical mechanism that bridges the gap between the LLM's reasoning capabilities and the vast array of external functionalities available.

此模式非常基礎，因為它突破 LLM 訓練資料的限制，讓模型能取得最新資訊、執行其內部不擅長的計算、存取使用者專屬資料，或觸發真實世界行動。函式呼叫是連結 LLM 推理能力與外部功能世界的技術橋樑。

While "function calling"aptly describes invoking specific, predefined code functions, it's useful to consider the more expansive concept of "tool calling."This broader term acknowledges that an agent's capabilities can extend far beyond simple function execution. A "tool"can be a traditional function, but it can also be a complex API endpoint, a request to a database, or even an instruction directed at another specialized agent. This perspective allows us to envision more sophisticated systems where, for instance, a primary agent might delegate a complex data analysis task to a dedicated "analyst agent"or query an external knowledge base through its API. Thinking in terms of "tool calling"better captures the full potential of agents to act as orchestrators across a diverse ecosystem of digital resources and other intelligent entities.

雖然「函式呼叫」能精準描述呼叫特定程式函式，但「工具呼叫」是更廣泛的概念。工具不只是一個函式，也可以是複雜的 API 端點、資料庫查詢，甚至是指派給其他專門代理人的指令。此視角讓我們能想像更進階的系統，例如主要代理人將複雜資料分析委派給「分析代理人」，或透過 API 查詢外部知識庫。以「工具呼叫」來思考，更能涵蓋代理人作為多元數位資源與智慧實體協調者的潛力。

Frameworks like LangChain, LangGraph, and Google Agent Developer Kit (ADK) provide robust support for defining tools and integrating them into agent workflows, often leveraging the native function calling capabilities of modern LLMs like those in the Gemini or OpenAI series. On the "canvas"of these frameworks, you define the tools and then configure agents (typically LLM Agents) to be aware of and capable of using these tools.

LangChain、LangGraph 與 Google Agent Developer Kit（ADK）等框架提供強大的工具定義與整合能力，常搭配 Gemini 或 OpenAI 系列等現代 LLM 的原生函式呼叫功能。在這些框架的「畫布」上，你先定義工具，再配置代理人（通常是 LLM 代理人）使其能理解並使用工具。

Tool Use is a cornerstone pattern for building powerful, interactive, and externally aware agents.

工具使用是打造強大、可互動且具外部感知的代理人的核心模式。

Practical Applications & Use Cases

實務應用與使用情境

The Tool Use pattern is applicable in virtually any scenario where an agent needs to go beyond generating text to perform an action or retrieve specific, dynamic information:

工具使用模式適用於幾乎所有需要代理人超越文字生成、執行動作或取得動態資訊的情境：

1. Information Retrieval from External Sources

1. 從外部來源取回資訊

Accessing real-time data or information that is not present in the LLM's training data.

取得 LLM 訓練資料中不存在的即時資訊。

- **Use Case:** A weather agent.
  - **Tool:** A weather API that takes a location and returns the current weather conditions.
  - **Agent Flow:** User asks, "What's the weather in London?", LLM identifies the need for the weather tool, calls the tool with "London", tool returns data, LLM formats the data into a user-friendly response.
- **使用情境：**天氣代理人。
  - **工具：**接收地點並回傳目前天氣狀況的天氣 API。

- **流程：**使用者問「倫敦天氣如何 ?」，LLM 判斷需要天氣工具，呼叫工具並傳入「London」，工具回傳資料，LLM 將資料整理成友善回應。

2. Interacting with Databases and APIs

2. 與資料庫與 API 互動

Performing queries, updates, or other operations on structured data.

對結構化資料進行查詢、更新或其他操作。

- **Use Case:** An e-commerce agent.
  - **Tools:** API calls to check product inventory, get order status, or process payments.
  - **Agent Flow:** User asks "Is product X in stock?", LLM calls the inventory API, tool returns stock count, LLM tells the user the stock status.
- **使用情境：**電商代理人。
  - **工具：**查詢庫存、訂單狀態或處理付款的 API。
  - **流程：**使用者問「X 商品有庫存嗎 ?」，LLM 呼叫庫存 API，工具回傳庫存數量，LLM 告知庫存狀態。

3. Performing Calculations and Data Analysis

3. 執行計算與資料分析

Using external calculators, data analysis libraries, or statistical tools.

使用外部計算器、資料分析函式庫或統計工具。

- **Use Case:** A financial agent.
  - **Tools:** A calculator function, a stock market data API, a spreadsheet tool.
  - **Agent Flow:** User asks "What's the current price of AAPL and calculate the potential profit if I bought 100 shares at $150?", LLM calls stock API, gets current price, then calls calculator tool, gets result, formats response.
- **使用情境：**金融代理人。

- **工具：**計算器函式、股市資料 API、試算表工具。

- **流程：**使用者問「AAPL 現價多少？若我以 150 美元買 100 股，潛在獲利是多少？」LLM 先呼叫股市 API 取得現價，再呼叫計算工具計算結果並回應。

4. Sending Communications

4. 發送訊息

Sending emails, messages, or making API calls to external communication services.

寄送電子郵件、訊息或呼叫外部通訊服務 API。

- **Use Case:** A personal assistant agent.
  - **Tool:** An email sending API.

  - **Agent Flow:** User says, “Send an email to John about the meeting tomorrow.”, LLM calls an email tool with the recipient, subject, and body extracted from the request.
- **使用情境：**個人助理代理人。
  - **工具：**發送郵件的 API。

  - **流程：**使用者說「寄信給 John 提醒明天會議」，LLM 以收件人、主旨與內文呼叫郵件工具。

5. Executing Code

5. 執行程式碼

Running code snippets in a safe environment to perform specific tasks.

在安全環境中執行程式碼片段以完成特定任務。

- **Use Case:** A coding assistant agent.
  - **Tool:** A code interpreter.

  - **Agent Flow:** User provides a Python snippet and asks, “What does this code do?”, LLM uses the interpreter tool to run the code and analyze its

output.

- **使用情境：**程式助理代理人。
  - **工具：**程式碼解譯器。
  - **流程：**使用者提供 Python 程式碼並問「這段在做什麼 ?」，LLM 使用解譯器執行並分析輸出。

6. Controlling Other Systems or Devices

6. 控制其他系統或裝置

Interacting with smart home devices, IoT platforms, or other connected systems.

與智慧家庭裝置、IoT 平台或其他連網系統互動。

- **Use Case:** A smart home agent.
  - **Tool:** An API to control smart lights.
  - **Agent Flow:** User says, “Turn off the living room lights.”LLM calls the smart home tool with the command and target device.
- **使用情境：**智慧家庭代理人。
  - **工具：**控制智慧燈具的 API。
  - **流程：**使用者說「關掉客廳燈」，LLM 呼叫智慧家庭工具傳入指令與目標裝置。

Tool Use is what transforms a language model from a text generator into an agent capable of sensing, reasoning, and acting in the digital or physical world (see Fig. 1)

工具使用讓語言模型從文字生成器轉變為能在數位或實體世界中感知、推理與行動的代理人（見圖 1）。

Fig.1: Some examples of an Agent using Tools

圖 1：代理人使用工具的例子

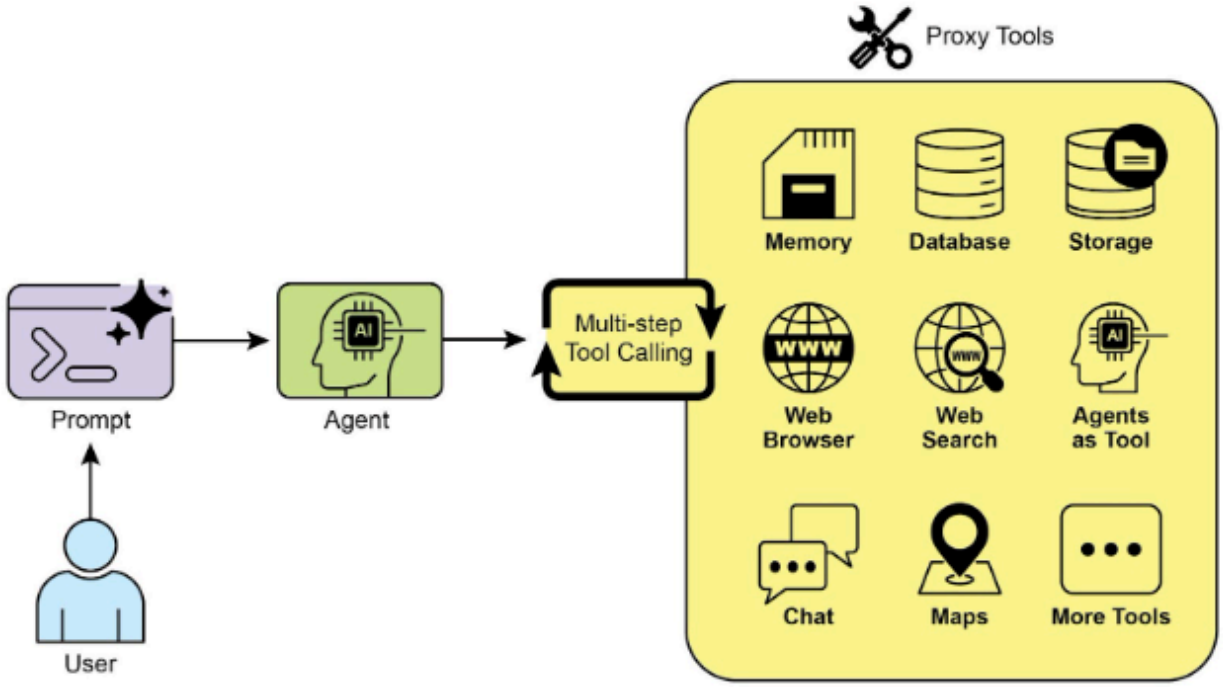

Figure 8: Some Examples of an Agent Using Tool

Hands–On Code Example (LangChain)

動手實作範例（LangChain）

The implementation of tool use within the LangChain framework is a two–stage process. Initially, one or more tools are defined, typically by encapsulating existing Python functions or other runnable components. Subsequently, these tools are bound to a language model, thereby granting the model the capability to generate a structured tool–use request when it determines that an external function call is required to fulfill a user's query.

在 LangChain 中實作工具使用通常分兩階段。首先定義一個或多個工具，通常是包裝現有 Python 函式或其他可執行元件。接著把這些工具綁定到語言模型，使模型在判斷需要外部函式時能產生結構化的工具呼叫請求。

The following implementation will demonstrate this principle by first defining a simple function to simulate an information retrieval tool. Following this, an agent will be constructed and configured to leverage this tool in response to user input. The execution of this example requires the installation of the core LangChain libraries and a model–specific provider package. Furthermore, proper authentication with the selected language model service, typically via an API key configured in the local environment, is a necessary prerequisite.

以下實作將先定義一個簡單函式來模擬資訊檢索工具，接著建構並配置代理人以在使用者輸入時使用該工具。執行此範例需要安裝 LangChain 核心套件與特定模型供應商套件，並在本機環境設定 API 金鑰進行驗證。

```
import os
import getpass
import asyncio
import nest_asyncio
from typing import List
from dotenv import load_dotenv
import logging

from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.prompts import ChatPromptTemplate
```

```python
from langchain_core.tools import tool as langchain_tool
from langchain.agents import create_tool_calling_agent, AgentExecutor


# UNCOMMENT
# Prompt the user securely and set API keys as environment variables
os.environ["GOOGLE_API_KEY"] = getpass.getpass("Enter your Google API key: ")
os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter your OpenAI API key: ")

try:
    # A model with function/tool calling capabilities is required.
    llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash", temperature=0)
    print(f" Language model initialized: {llm.model}")
except Exception as e:
    print(f" Error initializing language model: {e}")
    llm = None


# --- Define a Tool ---
@langchain_tool
def search_information(query: str) -> str:
    """
    Provides factual information on a given topic. Use this tool to find answers to ph
    like 'capital of France' or 'weather in London?'.
    """
    print(f"\n---    Tool Called: search_information with query: '{query}' ---")

    # Simulate a search tool with a dictionary of predefined results.
    simulated_results = {
        "weather in london": "The weather in London is currently cloudy with a temperatu
        "capital of france": "The capital of France is Paris.",
        "population of earth": "The estimated population of Earth is around 8 billion pe
        "tallest mountain": "Mount Everest is the tallest mountain above sea level.",
        "default": f"Simulated search result for '{query}': No specific information foun
```

```
    }
    result = simulated_results.get(query.lower(), simulated_results["default"])
    print(f"--- TOOL RESULT: {result} ---")
    return result


tools = [search_information]


# --- Create a Tool-Calling Agent ---
if llm:
    # This prompt template requires an `agent_scratchpad` placeholder for the agent's
    agent_prompt = ChatPromptTemplate.from_messages([
        ("system", "You are a helpful assistant."),
        ("human", "{input}"),
        ("placeholder", "{agent_scratchpad}"),
    ])

    # Create the agent, binding the LLM, tools, and prompt together.
    agent = create_tool_calling_agent(llm, tools, agent_prompt)

    # AgentExecutor is the runtime that invokes the agent and executes the chosen tool
    # The 'tools' argument is not needed here as they are already bound to the agent.
    agent_executor = AgentExecutor(agent=agent, verbose=True, tools=tools)


async def run_agent_with_tool(query: str):
    """Invokes the agent executor with a query and prints the final response."""
    print(f"\n---  Running Agent with Query: '{query}' ---")
    try:
        response = await agent_executor.ainvoke({"input": query})
        print("\n---  Final Agent Response ---")
        print(response["output"])
    except Exception as e:
```

```
        print(f"\n  An error occurred during agent execution: {e}")


async def main():
    """Runs all agent queries concurrently."""
    tasks = [
        run_agent_with_tool("What is the capital of France?"),
        run_agent_with_tool("What's the weather like in London?"),
        run_agent_with_tool("Tell me something about dogs."),  # Should trigger the def
    ]
    await asyncio.gather(*tasks)


nest_asyncio.apply()
asyncio.run(main())
```

The code sets up a tool–calling agent using the LangChain library and the Google Gemini model. It defines a `search_information` tool that simulates providing factual answers to specific queries. The tool has predefined responses for “weather in london,”“capital of france,”and “population of earth,”and a default response for other queries. A ChatGoogleGenerativeAI model is initialized, ensuring it has tool–calling capabilities. A ChatPromptTemplate is created to guide the agent’s interaction. The `create_tool_calling_agent` function is used to combine the language model, tools, and prompt into an agent. An AgentExecutor is then set up to manage the agent’s execution and tool invocation. The `run_agent_with_tool` asynchronous function is defined to invoke the agent with a given query and print the result. The main asynchronous function prepares multiple queries to be run concurrently. These queries are designed to test both the specific and default responses of the `search_information` tool. Finally, the asyncio.run(main()) call executes all the agent tasks. The code includes checks for successful LLM initialization before proceeding with agent setup and execution.

此程式碼使用 LangChain 與 Google Gemini 模型建立工具呼叫代理人，並定義 `search_information` 工具來模擬回答特定查詢。工具預先定義了「london 天氣」、「法國首都」、「地球人口」等回應，其他查詢則回傳預設回應。程式初始化 ChatGoogleGenerativeAI 模型以支援工具呼叫，並建立 ChatPromptTemplate 指引代

理人互動。create_tool_calling_agent 將模型、工具與提示組合成代理人，接著用 AgentExecutor 管理代理人執行與工具呼叫。run_agent_with_tool 是非同步函式，用來呼叫代理人並印出回應。主程式建立多個查詢並同時執行，以測試特定與預設回應。最後用 asyncio.run(main()) 執行所有任務。程式在開始前也檢查 LLM 是否成功初始化。

Hands-On Code Example (CrewAI)

動手實作範例（CrewAI）

This code provides a practical example of how to implement function calling (Tools) within the CrewAI framework. It sets up a simple scenario where an agent is equipped with a tool to look up information. The example specifically demonstrates fetching a simulated stock price using this agent and tool.

以下程式碼示範如何在 CrewAI 中實作函式呼叫（工具）。它建立一個簡單情境：代理人配備查詢工具，用於取得資訊。此範例以查詢模擬股價為例。

```
# pip install crewai langchain-openai

import os
from crewai import Agent, Task, Crew
from crewai.tools import tool
import logging


# --- Best Practice: Configure Logging ---
# A basic logging setup helps in debugging and tracking the crew's execution.
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)


# --- Set up your API Key ---
# For production, it's recommended to use a more secure method for key management
# like environment variables loaded at runtime or a secret manager.
#
# Set the environment variable for your chosen LLM provider (e.g., OPENAI_API_KEY)
```

```
# os.environ["OPENAI_API_KEY"] = "YOUR_API_KEY"
# os.environ["OPENAI_MODEL_NAME"] = "gpt-4o"


# --- 1. Refactored Tool: Returns Clean Data ---
# The tool now returns raw data (a float) or raises a standard Python error.
# This makes it more reusable and forces the agent to handle outcomes properly.
@tool("Stock Price Lookup Tool")
def get_stock_price(ticker: str) -> float:
    """
    Fetches the latest simulated stock price for a given stock ticker symbol.
    Returns the price as a float. Raises a ValueError if the ticker is not found.
    """
    logging.info(f"Tool Call: get_stock_price for ticker '{ticker}'")
    simulated_prices = {
        "AAPL": 178.15,
        "GOOGL": 1750.30,
        "MSFT": 425.50,
    }
    price = simulated_prices.get(ticker.upper())
    if price is not None:
        return price
    else:
        # Raising a specific error is better than returning a string.
        # The agent is equipped to handle exceptions and can decide on the next action
        raise ValueError(f"Simulated price for ticker '{ticker.upper()}' not found.")


# --- 2. Define the Agent ---
# The agent definition remains the same, but it will now leverage the improved tool.
financial_analyst_agent = Agent(
    role='Senior Financial Analyst',
    goal='Analyze stock data using provided tools and report key prices.',
    backstory="You are an experienced financial analyst adept at using data sources to f
```

```
    verbose=True,
    tools=[get_stock_price],
    # Allowing delegation can be useful, but is not necessary for this simple task.
    allow_delegation=False,
)


# --- 3. Refined Task: Clearer Instructions and Error Handling ---
# The task description is more specific and guides the agent on how to react
# to both successful data retrieval and potential errors.
analyze_aapl_task = Task(
    description=(
        "What is the current simulated stock price for Apple (ticker: AAPL)? "
        "Use the 'Stock Price Lookup Tool' to find it. "
        "If the ticker is not found, you must report that you were unable to retrieve th
    ),
    expected_output=(
        "A single, clear sentence stating the simulated stock price for AAPL. "
        "For example: 'The simulated stock price for AAPL is $178.15.' "
        "If the price cannot be found, state that clearly."
    ),
    agent=financial_analyst_agent,
)


# --- 4. Formulate the Crew ---
# The crew orchestrates how the agent and task work together.
financial_crew = Crew(
    agents=[financial_analyst_agent],
    tasks=[analyze_aapl_task],
    verbose=True  # Set to False for less detailed logs in production
)
```

```
# --- 5. Run the Crew within a Main Execution Block ---
# Using a __name__ == "__main__": block is a standard Python best practice.
def main():
    """Main function to run the crew."""
    # Check for API key before starting to avoid runtime errors.
    if not os.environ.get("OPENAI_API_KEY"):
        print("ERROR: The OPENAI_API_KEY environment variable is not set.")
        print("Please set it before running the script.")
        return

    print("\n## Starting the Financial Crew...")
    print("--------------------------------")

    # The kickoff method starts the execution.
    result = financial_crew.kickoff()

    print("\n--------------------------------")
    print("## Crew execution finished.")
    print("\nFinal Result:\n", result)


if __name__ == "__main__":
    main()
```

This code demonstrates a simple application using the Crew.ai library to simulate a financial analysis task. It defines a custom tool, `get_stock_price`, that simulates looking up stock prices for predefined tickers. The tool is designed to return a floating-point number for valid tickers or raise a ValueError for invalid ones. A Crew.ai Agent named `financial_analyst_agent` is created with the role of a Senior Financial Analyst. This agent is given the `get_stock_price` tool to interact with. A Task is defined, `analyze_aapl_task`, specifically instructing the agent to find the simulated stock price for AAPL using the tool. The task description includes clear instructions on how to handle both success and failure cases when using the tool. A Crew is assembled, comprising the `financial_analyst_agent` and the `analyze_aapl_task`. The verbose setting is enabled for both the agent and the crew to provide detailed logging dur-

ing execution. The main part of the script runs the crew's task using the kickoff() method within a standard `if __name__ == "__main__":` block. Before starting the crew, it checks if the `OPENAI_API_KEY` environment variable is set, which is required for the agent to function. The result of the crew's execution, which is the output of the task, is then printed to the console. The code also includes basic logging configuration for better tracking of the crew's actions and tool calls. It uses environment variables for API key management, though it notes that more secure methods are recommended for production environments. In short, the core logic showcases how to define tools, agents, and tasks to create a collaborative workflow in Crew.ai.

此程式碼示範使用 Crew.ai 模擬金融分析任務。它定義自訂工具 `get_stock_price`，用於查詢預先定義的股價。工具對有效代號回傳浮點數，無效代號則丟出 ValueError。接著建立名為 `financial_analyst_agent` 的代理人，角色為資深金融分析師，並賦予 `get_stock_price` 工具。定義任務 `analyze_aapl_task`，指示代理人用工具找出 AAPL 的模擬股價，並清楚說明成功與失敗情境的處理方式。再組裝 Crew，包含此代理人與任務，並開啟 verbose 以輸出詳盡紀錄。主程式以 kickoff() 執行，並在開始前檢查 `OPENAI_API_KEY` 是否設定。最後輸出結果。程式也設定了基本 logging，並提醒生產環境應使用更安全的金鑰管理。總結而言，核心流程展示了如何定義工具、代理人與任務以建立 Crew.ai 的協作流程。

Hands-on code (Google ADK)

動手實作（Google ADK）

The Google Agent Developer Kit (ADK) includes a library of natively integrated tools that can be directly incorporated into an agent's capabilities.

Google Agent Developer Kit（ADK）包含一組原生整合的工具庫，可直接納入代理人能力。

Google search: A primary example of such a component is the Google Search tool. This tool serves as a direct interface to the Google Search engine, equipping the agent with the functionality to perform web searches and retrieve external information.

Google search：其中一個重要元件是 Google Search 工具，作為 Google 搜尋引擎的直接介面，讓代理人能進行網頁搜尋並取得外部資訊。

```
from google.adk.agents import Agent as ADKAgent
```

```python
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.tools import google_search
from google.genai import types
import nest_asyncio
import asyncio


# Define variables required for Session setup and Agent execution
APP_NAME = "Google Search Agent"
USER_ID = "user1234"
SESSION_ID = "1234"


# Define Agent with access to search tool
root_agent = ADKAgent(
    name="basic_search_agent",
    model="gemini-2.0-flash-exp",
    description="Agent to answer questions using Google Search.",
    instruction="I can answer your questions by searching the internet. Just ask me anyt
    tools=[google_search],  # Google Search is a pre-built tool to perform Google searc
)


# Agent Interaction
async def call_agent(query: str):
    """
    Helper function to call the agent with a query.
    """
    # Session and Runner
    session_service = InMemorySessionService()
    await session_service.create_session(
        app_name=APP_NAME,
        user_id=USER_ID,
```

```
        session_id=SESSION_ID,
    )

    runner = Runner(agent=root_agent, app_name=APP_NAME, session_service=session_service

    content = types.Content(role='user', parts=[types.Part(text=query)])
    events = runner.run(user_id=USER_ID, session_id=SESSION_ID, new_message=content)

    for event in events:
        if event.is_final_response() and event.content:
            # Safely extract text from the final response
            if hasattr(event.content, "text") and event.content.text:
                final_response = event.content.text
            elif event.content.parts:
                final_response = "".join(
                    part.text for part in event.content.parts if getattr(part, "text", N
                )
            else:
                final_response = ""
            print("Agent Response:", final_response)


nest_asyncio.apply()
asyncio.run(call_agent("what's the latest ai news?"))
```

This code demonstrates how to create and use a basic agent powered by the Google ADK for Python. The agent is designed to answer questions by utilizing Google Search as a tool. First, necessary libraries from IPython, google.adk, and google.genai are imported. Constants for the application name, user ID, and session ID are defined. An Agent instance named `basic_search_agent` is created with a description and instructions indicating its purpose. It's configured to use the Google Search tool, which is a pre-built tool provided by the ADK. An InMemorySessionService (see Chapter 8) is initialized to manage sessions for the agent. A new session is created for the specified application, user, and session IDs. A Runner is instantiated, linking the created agent with the session service. This runner is responsible for executing the agent'

s interactions within a session. A helper function `call_agent` is defined to simplify the process of sending a query to the agent and processing the response. Inside `call_agent`, the user's query is formatted as a types.Content object with the role 'user'. The runner.run method is called with the user ID, session ID, and the new message content. The runner.run method returns a list of events representing the agent's actions and responses. The code iterates through these events to find the final response. If an event is identified as the final response, the text content of that response is extracted. The extracted agent response is then printed to the console. Finally, the `call_agent` function is called with the query "what's the latest ai news?" to demonstrate the agent in action.

此程式碼示範如何使用 Google ADK 建立基本代理人並呼叫 Google Search 工具回答問題。程式先匯入必要套件，定義應用名稱、使用者 ID 與 Session ID。建立 `basic_search_agent` 並配置 Google Search 工具。使用 InMemorySessionService 管理 session，建立 session 後以 Runner 執行代理人互動。`call_agent` 以 types.Content 格式送出使用者查詢，runner.run 會回傳事件列表，程式從最終回應事件取出文字並印出。最後以「what's the latest ai news?」作為示範查詢。

Code execution: The Google ADK features integrated components for specialized tasks, including an environment for dynamic code execution. The `built_in_code_execution` tool provides an agent with a sandboxed Python interpreter. This allows the model to write and run code to perform computational tasks, manipulate data structures, and execute procedural scripts. Such functionality is critical for addressing problems that require deterministic logic and precise calculations, which are outside the scope of probabilistic language generation alone.

程式碼執行： Google ADK 提供用於專門任務的整合元件，包括動態程式碼執行環境。`built_in_code_execution` 工具提供沙盒化 Python 直譯器，讓模型可撰寫並執行程式碼，以完成計算、操作資料結構或執行程序腳本。此功能對需要確定性邏輯與精準計算的問題至關重要，因為這超出單純機率式語言生成的範圍。

```
import os
import getpass
import asyncio
import nest_asyncio
from typing import List
from dotenv import load_dotenv
```

```python
import logging

from google.adk.agents import Agent as ADKAgent, LlmAgent
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService
from google.adk.tools import google_search
from google.adk.code_executors import BuiltInCodeExecutor
from google.genai import types


# Define variables required for Session setup and Agent execution
APP_NAME = "calculator"
USER_ID = "user1234"
SESSION_ID = "session_code_exec_async"


# Agent Definition
code_agent = LlmAgent(
    name="calculator_agent",
    model="gemini-2.0-flash",
    code_executor=BuiltInCodeExecutor(),
    instruction="""You are a calculator agent.
    When given a mathematical expression, write and execute Python code to calculate the
    Return only the final numerical result as plain text, without markdown or code block
    """,
    description="Executes Python code to perform calculations.",
)


# Agent Interaction (Async)
async def call_agent_async(query: str):
    # Session and Runner
    session_service = InMemorySessionService()
    await session_service.create_session(app_name=APP_NAME, user_id=USER_ID, session_id=
```

```
runner = Runner(agent=code_agent, app_name=APP_NAME, session_service=session_service

content = types.Content(role='user', parts=[types.Part(text=query)])
print(f"\n--- Running Query: {query} ---")

try:
    # Use run_async
    async for event in runner.run_async(user_id=USER_ID, session_id=SESSION_ID, new_
        print(f"Event ID: {event.id}, Author: {event.author}")

        if event.content and event.content.parts and event.is_final_response():
            for part in event.content.parts:  # Iterate through all parts
                if getattr(part, "executable_code", None):
                    # Access the actual code string via .code
                    print(f"  Debug: Agent generated code:\n```python\n{part.executa
                elif getattr(part, "code_execution_result", None):
                    # Access outcome and output correctly
                    print(
                        "  Debug: Code Execution Result: "
                        f"{part.code_execution_result.outcome} - Output:\n{part.code
                    )
                elif getattr(part, "text", None) and not part.text.isspace():
                    # Also print any text parts found in any event for debugging
                    print(f"  Text: '{part.text.strip()}'")

            # --- Check for final response AFTER specific parts ---
            text_parts = [part.text for part in event.content.parts if getattr(part,
            final_result = "".join(text_parts)
            print(f"==> Final Agent Response: {final_result}")

except Exception as e:
    print(f"ERROR during agent run: {e}")
```

```
    print("-" * 30)


# Main async function to run the examples
async def main():
    await call_agent_async("Calculate the value of (5 + 7) * 3")
    await call_agent_async("What is 10 factorial?")


# Execute the main async function
try:
    nest_asyncio.apply()
    asyncio.run(main())
except RuntimeError as e:
    # Handle specific error when running asyncio.run in an already running loop (like
    if "cannot be called from a running event loop" in str(e):
        print("\nRunning in an existing event loop (like Colab/Jupyter).")
        print("Please run `await main()` in a notebook cell instead.")
        # If in an interactive environment like a notebook, you might need to run:
        # await main()
    else:
        raise e  # Re-raise other runtime errors
```

This script uses Google's Agent Development Kit (ADK) to create an agent that solves mathematical problems by writing and executing Python code. It defines an LlmAgent specifically instructed to act as a calculator, equipping it with the `built_in_code_execution` tool. The primary logic resides in the `call_agent_async` function, which sends a user's query to the agent's runner and processes the resulting events. Inside this function, an asynchronous loop iterates through events, printing the generated Python code and its execution result for debugging. The code carefully distinguishes between these intermediate steps and the final event containing the numerical answer. Finally, a main function runs the agent with two different mathematical expressions to demonstrate its ability to perform calculations.

此腳本使用 Google ADK 建立一個能以 Python 寫碼並執行計算的代理人。它定義一個 LlmAgent 作為計算器，並配備 `built_in_code_execution` 工具。主要流程在

call_agent_async，它將使用者查詢送入 Runner，並處理回傳事件。在此函式中，非同步迴圈遍歷事件，印出代理人生成的 Python 程式碼與執行結果以供除錯。程式清楚區分中間步驟與最終數值回應。最後主函式以兩個數學表達式示範其計算能力。

Enterprise search: This code defines a Google ADK application using the google.adk library in Python. It specifically uses a VSearchAgent, which is designed to answer questions by searching a specified Vertex AI Search datastore. The code initializes a VSearchAgent named q2_strategy_vsearch_agent, providing a description, the model to use ("gemini-2.0-flash-exp"), and the ID of the Vertex AI Search datastore. The DATASTORE_ID is expected to be set as an environment variable. It then sets up a Runner for the agent, using an InMemorySessionService to manage conversation history. An asynchronous function call_vsearch_agent_async is defined to interact with the agent. This function takes a query, constructs a message content object, and calls the runner's run_async method to send the query to the agent. The function then streams the agent's response back to the console as it arrives. It also prints information about the final response, including any source attributions from the datastore. Error handling is included to catch exceptions during the agent's execution, providing informative messages about potential issues like an incorrect datastore ID or missing permissions. Another asynchronous function run_vsearch_example is provided to demonstrate how to call the agent with example queries. The main execution block checks if the DATASTORE_ID is set and then runs the example using asyncio.run. It includes a check to handle cases where the code is run in an environment that already has a running event loop, like a Jupyter notebook.

企業搜尋：這段程式碼使用 google.adk 套件建立 Google ADK 應用，並使用 VSearchAgent 透過指定的 Vertex AI Search datastore 回答問題。程式初始化 q2_strategy_vsearch_agent，指定描述、模型 ("gemini-2.0-flash-exp") 與 datastore ID。DATASTORE_ID 需在環境變數中設定。接著以 InMemorySessionService 管理對話歷史並建立 Runner。call_vsearch_agent_async 非同步函式用來發送查詢：建立訊息內容後呼叫 runner.run_async，並串流輸出回應，同時列出來源引用。程式包含錯誤處理，提示可能是 datastore ID 錯誤或權限不足。run_vsearch_example 提供範例查詢，主程式在確認 DATASTORE_ID 存在後執行範例，也處理在 Jupyter 等已有事件迴圈環境下的情況。

```
import asyncio
import os
```

```python
from google.genai import types
from google.adk import agents
from google.adk.runners import Runner
from google.adk.sessions import InMemorySessionService


# --- Configuration ---
# Ensure you have set your GOOGLE_API_KEY and DATASTORE_ID environment variables
# For example:
# os.environ["GOOGLE_API_KEY"] = "YOUR_API_KEY"
# os.environ["DATASTORE_ID"] = "YOUR_DATASTORE_ID"
DATASTORE_ID = os.environ.get("DATASTORE_ID")


# --- Application Constants ---
APP_NAME = "vsearch_app"
USER_ID = "user_123"  # Example User ID
SESSION_ID = "session_456"  # Example Session ID


# --- Agent Definition (Updated with the newer model from the guide) ---
vsearch_agent = agents.VSearchAgent(
    name="q2_strategy_vsearch_agent",
    description="Answers questions about Q2 strategy documents using Vertex AI Search.",
    model="gemini-2.0-flash-exp",  # Updated model based on the guide's examples
    datastore_id=DATASTORE_ID,
    model_parameters={"temperature": 0.0},
)


# --- Runner and Session Initialization ---
runner = Runner(
    agent=vsearch_agent,
```

```
    app_name=APP_NAME,
    session_service=InMemorySessionService(),
)


# --- Agent Invocation Logic ---
async def call_vsearch_agent_async(query: str):
    """Initializes a session and streams the agent's response."""
    print(f"User: {query}")
    print("Agent: ", end="", flush=True)
    try:
        # Construct the message content correctly
        content = types.Content(role='user', parts=[types.Part(text=query)])

        # Process events as they arrive from the asynchronous runner
        async for event in runner.run_async(
            user_id=USER_ID,
            session_id=SESSION_ID,
            new_message=content,
        ):
            # For token-by-token streaming of the response text
            if hasattr(event, "content_part_delta") and event.content_part_delta:
                print(event.content_part_delta.text, end="", flush=True)

            # Process the final response and its associated metadata
            if event.is_final_response():
                print()  # Newline after the streaming response
                if getattr(event, "grounding_metadata", None):
                    print(
                        f"  (Source Attributions: "
                        f"{len(event.grounding_metadata.grounding_attributions)} sources
                    )
                else:
                    print("  (No grounding metadata found)")
```

```
                print("-" * 30)
    except Exception as e:
        print(f"\nAn error occurred: {e}")
        print("Please ensure your datastore ID is correct and that the service account h
        print("-" * 30)


# --- Run Example ---
async def run_vsearch_example():
    # Replace with a question relevant to YOUR datastore content
    await call_vsearch_agent_async("Summarize the main points about the Q2 strategy docu
    await call_vsearch_agent_async("What safety procedures are mentioned for lab X?")


# --- Execution ---
if __name__ == "__main__":
    if not DATASTORE_ID:
        print("Error: DATASTORE_ID environment variable is not set.")
    else:
        try:
            asyncio.run(run_vsearch_example())
        except RuntimeError as e:
            # This handles cases where asyncio.run is called in an environment
            # that already has a running event loop (like a Jupyter notebook).
            if "cannot be called from a running event loop" in str(e):
                print("Skipping execution in a running event loop. Please run this scrip
            else:
                raise e
```

Overall, this code provides a basic framework for building a conversational AI application that leverages Vertex AI Search to answer questions based on information stored in a datastore. It demonstrates how to define an agent, set up a runner, and interact with the agent asynchronously while streaming the response. The focus is on retrieving and synthesizing information from a specific datastore to answer user queries.

總結來說，這段程式提供一個基本框架，用於建立利用 Vertex AI Search 的對話式 AI 應用，從 datastore 中檢索並整合資訊來回答問題。它示範如何定義代理人、設定 Runner，並以非同步方式串流回應與代理人互動，聚焦於從特定資料庫取得與整合資訊。

Vertex Extensions: A Vertex AI extension is a structured API wrapper that enables a model to connect with external APIs for real-time data processing and action execution. Extensions offer enterprise-grade security, data privacy, and performance guarantees. They can be used for tasks like generating and running code, querying websites, and analyzing information from private datastores. Google provides prebuilt extensions for common use cases like Code Interpreter and Vertex AI Search, with the option to create custom ones. The primary benefit of extensions includes strong enterprise controls and seamless integration with other Google products. The key difference between extensions and function calling lies in their execution: Vertex AI automatically executes extensions, whereas function calls require manual execution by the user or client.

Vertex Extensions： Vertex AI 擴充功能是一種結構化 API 包裝器，讓模型能連接外部 API 以進行即時資料處理與行動執行。擴充功能提供企業級安全性、資料隱私與效能保證，可用於生成並執行程式碼、查詢網站或分析私有資料庫資訊。Google 提供 Code Interpreter、Vertex AI Search 等常見用例的預建擴充，也可自訂。擴充功能的主要優勢是強大的企業控制與與 Google 產品的無縫整合。擴充與函式呼叫的關鍵差異在於執行方式：Vertex AI 會自動執行擴充，而函式呼叫需由使用者或客戶端手動執行。

At a Glance

一覽

What: LLMs are powerful text generators, but they are fundamentally disconnected from the outside world. Their knowledge is static, limited to the data they were trained on, and they lack the ability to perform actions or retrieve real-time information. This inherent limitation prevents them from completing tasks that require interaction with external APIs, databases, or services. Without a bridge to these external systems, their utility for solving real-world problems is severely constrained.

是什麼： LLM 雖是強大的文字生成器，但本質上與外部世界隔離。其知識是靜態的，僅限訓練資料，缺乏執行動作或取得即時資訊的能力。這個根本限制使其無法完成需要與外

部 API、資料庫或服務互動的任務。缺乏外部系統的橋樑，其解決真實世界問題的效用便大幅受限。

Why: The Tool Use pattern, often implemented via function calling, provides a standardized solution to this problem. It works by describing available external functions, or "tools,"to the LLM in a way it can understand. Based on a user's request, the agentic LLM can then decide if a tool is needed and generate a structured data object (like a JSON) specifying which function to call and with what arguments. An orchestration layer executes this function call, retrieves the result, and feeds it back to the LLM. This allows the LLM to incorporate up-to-date, external information or the result of an action into its final response, effectively giving it the ability to act.

為什麼：工具使用模式（常透過函式呼叫實作）提供標準化解法。它先以 LLM 可理解的方式描述可用外部函式/工具。代理式 LLM 會依使用者需求判斷是否需使用工具，並生成結構化資料（如 JSON）指定呼叫哪個函式與參數。編排層執行呼叫並回傳結果給 LLM，讓 LLM 能將最新外部資訊或執行結果納入最終回應，等同賦予其行動能力。

Rule of thumb: Use the Tool Use pattern whenever an agent needs to break out of the LLM's internal knowledge and interact with the outside world. This is essential for tasks requiring real-time data (e.g., checking weather, stock prices), accessing private or proprietary information (e.g., querying a company's database), performing precise calculations, executing code, or triggering actions in other systems (e.g., sending an email, controlling smart devices).

經驗法則：當代理人需要跳出 LLM 內部知識並與外部世界互動時，就應使用工具使用模式。這對需要即時資料（如天氣、股價）、存取私有或專有資訊（如公司資料庫）、精準計算、執行程式碼或觸發其他系統行動（如寄信、控制智慧裝置）的任務不可或缺。

Visual summary:

視覺摘要：

Fig.2: Tool use design pattern

圖 2：工具使用設計模式

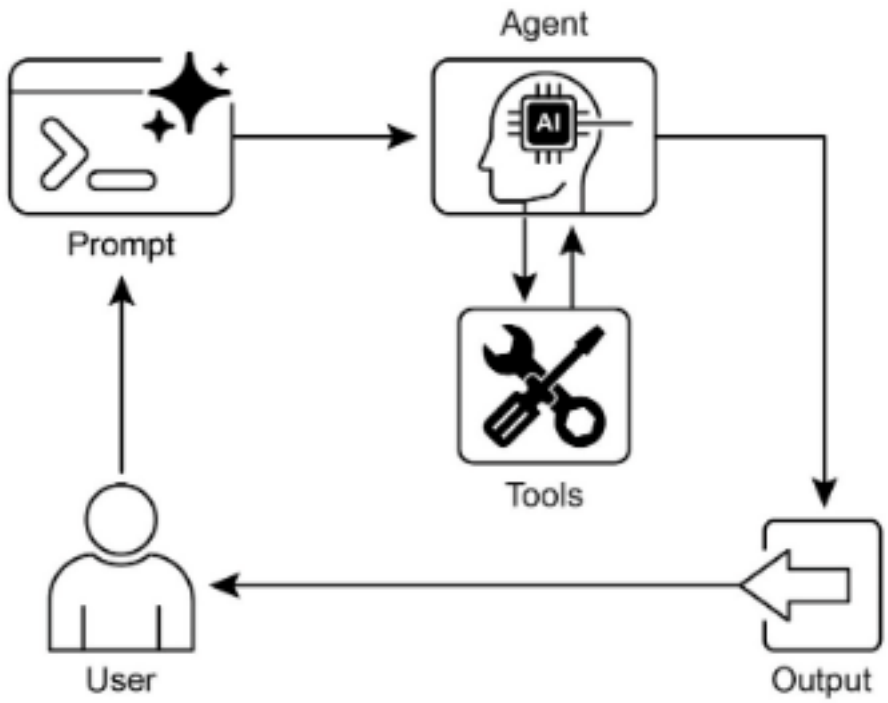

Figure 9: Tool Use Design Pattern

Key Takeaways

重點整理

- Tool Use (Function Calling) allows agents to interact with external systems and access dynamic information.
- It involves defining tools with clear descriptions and parameters that the LLM can understand.
- The LLM decides when to use a tool and generates structured function calls.
- Agentic frameworks execute the actual tool calls and return the results to the LLM.
- Tool Use is essential for building agents that can perform real-world actions

and provide up-to-date information.

- LangChain simplifies tool definition using the @tool decorator and provides `create_tool_calling_agent` and AgentExecutor for building tool-using agents.

- Google ADK has a number of very useful pre-built tools such as Google Search, Code Execution and Vertex AI Search Tool.
- 工具使用（函式呼叫）讓代理人能與外部系統互動並取得動態資訊。

- 需要以清楚描述與參數定義工具，使 LLM 能理解。

- LLM 會決定何時使用工具並產生結構化呼叫。

- 代理框架負責實際執行工具並回傳結果給 LLM。

- 工具使用是建構能執行真實世界行動並提供最新資訊之代理人的必要條件。

- LangChain 透過 @tool 簡化工具定義，並提供 `create_tool_calling_agent` 與 AgentExecutor 建立工具型代理人。

- Google ADK 提供多種預建工具，如 Google Search、Code Execution 與 Vertex AI Search。

Conclusion

結論

The Tool Use pattern is a critical architectural principle for extending the functional scope of large language models beyond their intrinsic text generation capabilities. By equipping a model with the ability to interface with external software and data sources, this paradigm allows an agent to perform actions, execute computations,

and retrieve information from other systems. This process involves the model generating a structured request to call an external tool when it determines that doing so is necessary to fulfill a user's query. Frameworks such as LangChain, Google ADK, and Crew AI offer structured abstractions and components that facilitate the integration of these external tools. These frameworks manage the process of exposing tool specifications to the model and parsing its subsequent tool-use requests. This simplifies the development of sophisticated agentic systems that can interact with and take action within external digital environments.

工具使用模式是擴展大型語言模型功能範圍的重要架構原則，讓模型超越文字生成的內在能力。透過讓模型能與外部軟體與資料來源介接，代理人得以執行動作、進行計算並從其他系統取得資訊。此流程包含模型在需要時生成結構化工具呼叫請求。LangChain、Google ADK 與 Crew AI 等框架提供結構化抽象與元件來整合外部工具，並管理工具規格的揭露與工具請求的解析，使開發能在外部數位環境中互動與採取行動的進階代理系統變得更簡化。

References

參考資料

1. LangChain Documentation (Tools): https://python.langchain.com/docs/integrations/tools/
2. Google Agent Developer Kit (ADK) Documentation (Tools): https://google.github.io/adk-docs/tools/
3. OpenAI Function Calling Documentation: https://platform.openai.com/docs/guides/function-calling
4. CrewAI Documentation (Tools): https://docs.crewai.com/concepts/tools

Chapter 6: Planning

第 6 章：規劃

Intelligent behavior often involves more than just reacting to the immediate input. It requires foresight, breaking down complex tasks into smaller, manageable steps, and strategizing how to achieve a desired outcome. This is where the Planning pattern comes into play. At its core, planning is the ability for an agent or a system of agents to formulate a sequence of actions to move from an initial state towards a goal state.

智慧行為往往不只是對即時輸入做反應，而是需要前瞻性：將複雜任務拆解為可管理的小步驟，並制定達成目標的策略。這就是規劃模式的角色。本質上，規劃是讓代理人或代理人系統能制定一連串行動，從初始狀態推進到目標狀態的能力。

Planning Pattern Overview

規劃模式概覽

In the context of AI, it's helpful to think of a planning agent as a specialist to whom you delegate a complex goal. When you ask it to "organize a team offsite," you are defining the what—the objective and its constraints—but not the how. The agent's core task is to autonomously chart a course to that goal. It must first understand the initial state (e.g., budget, number of participants, desired dates) and the goal state (a successfully booked offsite), and then discover the optimal sequence of actions to connect them. The plan is not known in advance; it is created in response to the request.

在 AI 的脈絡中，可把規劃代理人視為你委派複雜目標的專家。當你要求它「規劃團隊外出活動」時，你定義的是「做什麼」——目標與限制，而不是「怎麼做」。代理人的核心任務是自主繪製通往目標的路徑。它必須先理解初始狀態（如預算、人數、日期）與目標狀態（成功訂好活動），再找出連結兩者的最佳行動序列。計畫不是預先存在的，而是因應需求而產生。

A hallmark of this process is adaptability. An initial plan is merely a starting point, not a rigid script. The agent's real power is its ability to incorporate new information and steer the project around obstacles. For instance, if the preferred venue becomes

unavailable or a chosen caterer is fully booked, a capable agent doesn't simply fail. It adapts. It registers the new constraint, re-evaluates its options, and formulates a new plan, perhaps by suggesting alternative venues or dates.

此流程的特徵之一是適應力。初始計畫只是起點，而非僵化腳本。代理人的真正力量在於能納入新資訊並繞過障礙。例如首選場地不可用或餐飲已滿，優秀的代理人不會直接失敗，而是調整策略：記錄新限制、重新評估選項，並提出新計畫，例如替代場地或日期。

However, it is crucial to recognize the trade-off between flexibility and predictability. Dynamic planning is a specific tool, not a universal solution. When a problem's solution is already well-understood and repeatable, constraining the agent to a predetermined, fixed workflow is more effective. This approach limits the agent's autonomy to reduce uncertainty and the risk of unpredictable behavior, guaranteeing a reliable and consistent outcome. Therefore, the decision to use a planning agent versus a simple task-execution agent hinges on a single question: does the "how" need to be discovered, or is it already known?

然而，必須理解彈性與可預測性之間的取捨。動態規劃是一種特定工具，並非萬用解。當問題解法已明確且可重複時，將代理人限制於預先固定的流程反而更有效。這種方式降低代理人的自主性，以減少不確定性與不可預測行為風險，確保穩定一致的結果。因此，是否使用規劃代理人或單純任務執行代理人，關鍵在於：解法的「怎麼做」是否需要被發現，或早已知曉？

Practical Applications & Use Cases

實務應用與使用情境

The Planning pattern is a core computational process in autonomous systems, enabling an agent to synthesize a sequence of actions to achieve a specified goal, particularly within dynamic or complex environments. This process transforms a high-level objective into a structured plan composed of discrete, executable steps.

規劃模式是自主系統中的核心計算流程，使代理人能在動態或複雜環境中綜合出行動序列以達成特定目標。此流程會把高階目標轉化為由離散、可執行步驟構成的結構化計畫。

In domains such as procedural task automation, planning is used to orchestrate complex workflows. For example, a business process like onboarding a new employee can be decomposed into a directed sequence of sub-tasks, such as creating system

accounts, assigning training modules, and coordinating with different departments. The agent generates a plan to execute these steps in a logical order, invoking necessary tools or interacting with various systems to manage dependencies.

在程序式任務自動化等領域，規劃用於協調複雜工作流。例如新員工入職流程可拆成一系列子任務：建立系統帳號、指派訓練模組、與各部門協調等。代理人會生成計畫，按邏輯順序執行步驟，並呼叫必要工具或與各系統互動以管理相依關係。

Within robotics and autonomous navigation, planning is fundamental for state-space traversal. A system, whether a physical robot or a virtual entity, must generate a path or sequence of actions to transition from an initial state to a goal state. This involves optimizing for metrics such as time or energy consumption while adhering to environmental constraints, like avoiding obstacles or following traffic regulations.

在機器人與自主導航中，規劃是進行狀態空間移動的基礎。無論是實體機器人或虛擬實體，都必須生成路徑或行動序列，從初始狀態走到目標狀態。這涉及在時間或能耗等指標上做最佳化，同時遵循環境限制，例如避開障礙或遵守交通規則。

This pattern is also critical for structured information synthesis. When tasked with generating a complex output like a research report, an agent can formulate a plan that includes distinct phases for information gathering, data summarization, content structuring, and iterative refinement. Similarly, in customer support scenarios involving multi-step problem resolution, an agent can create and follow a systematic plan for diagnosis, solution implementation, and escalation.

此模式也對結構化資訊整合至關重要。當需要產出複雜成果（如研究報告）時，代理人可制定包含資訊蒐集、資料摘要、內容結構化與迭代精煉的計畫。同樣地，在需要多步驟問題解決的客服情境中，代理人可建立並執行診斷、解法實施與升級處理的系統化計畫。

In essence, the Planning pattern allows an agent to move beyond simple, reactive actions to goal-oriented behavior. It provides the logical framework necessary to solve problems that require a coherent sequence of interdependent operations.

總結而言，規劃模式讓代理人超越單純的反應式行為，轉向目標導向行動，並提供解決需多步驟相依操作問題的邏輯框架。

Hands-on code (Crew AI)

動手實作（Crew AI）

The following section will demonstrate an implementation of the Planner pattern using the Crew AI framework. This pattern involves an agent that first formulates a multi-step plan to address a complex query and then executes that plan sequentially.

以下示範使用 Crew AI 框架實作規劃者模式。此模式讓代理人先為複雜問題制定多步驟計畫，再依序執行。

```
import os
from dotenv import load_dotenv
from crewai import Agent, Task, Crew, Process
from langchain_openai import ChatOpenAI


# Load environment variables from .env file for security
load_dotenv()


# 1. Explicitly define the language model for clarity
llm = ChatOpenAI(model="gpt-4-turbo")


# 2. Define a clear and focused agent
planner_writer_agent = Agent(
    role='Article Planner and Writer',
    goal='Plan and then write a concise, engaging summary on a specified topic.',
    backstory=(
        'You are an expert technical writer and content strategist. '
        'Your strength lies in creating a clear, actionable plan before writing, '
        'ensuring the final summary is both informative and easy to digest.'
    ),
    verbose=True,
    allow_delegation=False,
```

```
    llm=llm,  # Assign the specific LLM to the agent
)


# 3. Define a task with a more structured and specific expected output
topic = "The importance of Reinforcement Learning in AI"

high_level_task = Task(
    description=(
        f"1. Create a bullet-point plan for a summary on the topic: '{topic}'.\n"
        f"2. Write the summary based on your plan, keeping it around 200 words."
    ),
    expected_output=(
        "A final report containing two distinct sections:\n\n"
        "### Plan\n"
        "- A bulleted list outlining the main points of the summary.\n\n"
        "### Summary\n"
        "- A concise and well-structured summary of the topic."
    ),
    agent=planner_writer_agent,
)


# Create the crew with a clear process
crew = Crew(
    agents=[planner_writer_agent],
    tasks=[high_level_task],
    process=Process.sequential,
)


# Execute the task
print("## Running the planning and writing task ##")
result = crew.kickoff()
```

```
print("\n\n---\n## Task Result ##\n---")
print(result)
```

This code uses the CrewAI library to create an AI agent that plans and writes a summary on a given topic. It starts by importing necessary libraries, including Crew.ai and `langchain_openai`, and loading environment variables from a .env file. A ChatOpenAI language model is explicitly defined for use with the agent. An Agent named `planner_writer_agent` is created with a specific role and goal: to plan and then write a concise summary. The agent's backstory emphasizes its expertise in planning and technical writing. A Task is defined with a clear description to first create a plan and then write a summary on the topic "The importance of Reinforcement Learning in AI", with a specific format for the expected output. A Crew is assembled with the agent and task, set to process them sequentially. Finally, the crew.kickoff() method is called to execute the defined task and the result is printed.

此程式碼使用 CrewAI 建立一個 AI 代理人，用於規劃並撰寫指定主題摘要。它先匯入必要套件並從.env 載入環境變數，明確指定 ChatOpenAI 模型。建立 `planner_writer_agent`，角色與目標是先規劃再撰寫精簡摘要，並以其背景說明強調規劃與技術寫作專長。接著定義任務：先對「強化學習在 AI 中的重要性」產生要點計畫，再據此撰寫約 200 字摘要，且指定預期輸出格式。最後組裝 Crew，採序列流程執行，並輸出結果。

Google DeepResearch

Google DeepResearch

Google Gemini DeepResearch (see Fig.1) is an agent-based system designed for autonomous information retrieval and synthesis. It functions through a multi-step agentic pipeline that dynamically and iteratively queries Google Search to systematically explore complex topics. The system is engineered to process a large corpus of web-based sources, evaluate the collected data for relevance and knowledge gaps, and perform subsequent searches to address them. The final output consolidates the vetted information into a structured, multi-page summary with citations to the original sources.

Google Gemini DeepResearch（見圖 1）是一個以代理人為核心的自主資訊檢索與整合系統。它透過多步驟代理管線，動態且迭代地查詢 Google Search，以系統化探索複雜主題。系統設計可處理大量網頁來源、評估資料的相關性與知識缺口，並進一步搜尋補足。最終輸出將經過篩選的資訊整合為具引用的多頁結構化摘要。

Expanding on this, the system's operation is not a single query–response event but a managed, long–running process. It begins by deconstructing a user's prompt into a multi–point research plan (see Fig. 1), which is then presented to the user for review and modification. This allows for a collaborative shaping of the research trajectory before execution. Once the plan is approved, the agentic pipeline initiates its iterative search–and–analysis loop. This involves more than just executing a series of predefined searches; the agent dynamically formulates and refines its queries based on the information it gathers, actively identifying knowledge gaps, corroborating data points, and resolving discrepancies.

進一步來看，系統的運作不是單次問答事件，而是受控的長時間流程。它先將使用者提示拆解為多點研究計畫（見圖 1），再提供給使用者檢視與修訂，以便在執行前共同塑造研究路徑。一旦計畫核准，代理管線就會啟動反覆的搜尋與分析迴圈。這不僅是執行預先定義的搜尋，而是依據蒐集到的資訊動態調整查詢、辨識知識缺口、驗證資料點並解決矛盾。

Fig. 1: Google Deep Research agent generating an execution plan for using Google Search as a tool.

圖 1：Google Deep Research 代理人生成使用 Google Search 工具的執行計畫。

A key architectural component is the system's ability to manage this process asynchronously. This design ensures that the investigation, which can involve analyzing hundreds of sources, is resilient to single–point failures and allows the user to disengage and be notified upon completion. The system can also integrate user–provided documents, combining information from private sources with its web–based research. The final output is not merely a concatenated list of findings but a structured, multi–page report. During the synthesis phase, the model performs a critical evaluation of the collected information, identifying major themes and organizing the content into a coherent narrative with logical sections. The report is designed to be interactive, often including features like an audio overview, charts, and links to the original cited sources, allowing for verification and further exploration by the user. In addition to the synthesized results, the model explicitly returns the full list of sources it searched and

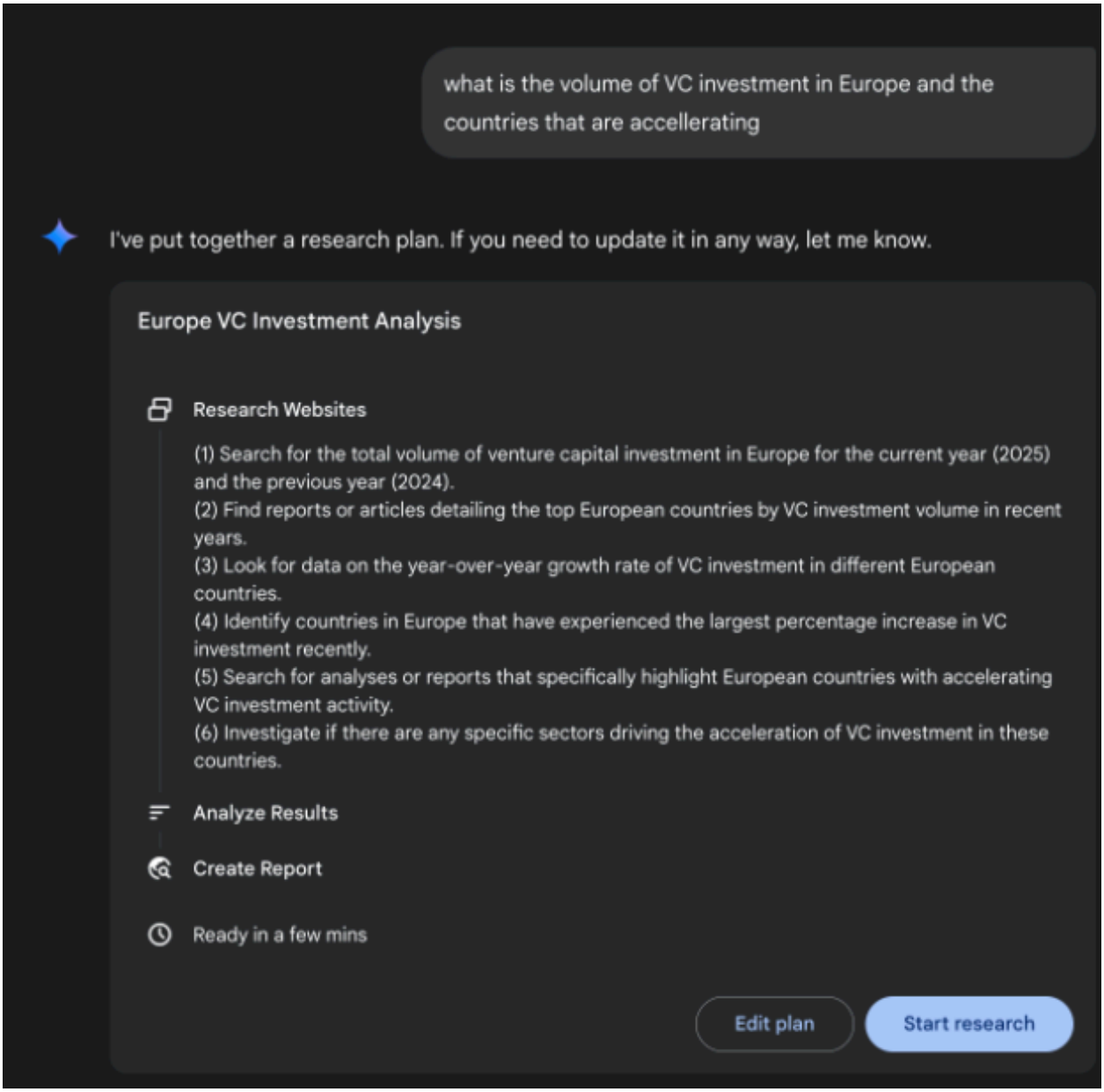

Figure 10: Google Deep Research agent generating an execution plan for using Google Search as a tool

consulted (see Fig.2). These are presented as citations, providing complete transparency and direct access to the primary information. This entire process transforms a simple query into a comprehensive, synthesized body of knowledge.

系統的一個關鍵架構能力是能以非同步方式管理此流程。這種設計讓可能分析數百來源的調查能抗單點失效，並允許使用者中途離開、完成後再通知。系統也可整合使用者提供文件，將私有資訊與網路研究結合。最終輸出不只是把結果串接，而是多頁結構化報告。綜合階段中，模型會對收集資訊做批判性評估、辨識主要主題並以合理章節組織成連貫敘事。報告設計為可互動，常包含語音概覽、圖表與原始來源連結，方便使用者驗證與深入探索。除整合結果外，模型也會回傳完整搜尋與引用來源清單（見圖 2），以引用形式呈現，提供完整透明度與對原始資訊的直接存取。整體流程把簡單查詢轉化為完整、綜合的知識體。

Fig. 2: An example of Deep Research plan being executed, resulting in Google Search being used as a tool to search various web sources.

圖 2：深度研究計畫執行示例，使用 Google Search 工具搜尋多個網路來源。

By mitigating the substantial time and resource investment required for manual data acquisition and synthesis, Gemini DeepResearch provides a more structured and exhaustive method for information discovery. The system's value is particularly evident in complex, multi-faceted research tasks across various domains.

透過降低人工蒐集與整合資料所需的大量時間與資源，Gemini DeepResearch 提供更結構化、全面的資訊探索方式。其價值在跨領域的複雜研究任務中特別顯著。

For instance, in competitive analysis, the agent can be directed to systematically gather and collate data on market trends, competitor product specifications, public sentiment from diverse online sources, and marketing strategies. This automated process replaces the laborious task of manually tracking multiple competitors, allowing analysts to focus on higher-order strategic interpretation rather than data collection (see Fig. 3).

例如在競品分析中，代理人可被指示系統性蒐集市場趨勢、競品規格、各種網路來源的公眾情緒與行銷策略。這個自動化流程取代了手動追蹤多家競品的繁重工作，使分析師能聚焦於更高層次的策略解讀，而非資料蒐集（見圖 3）。

Fig. 3: Final output generated by the Google Deep Research agent, analyzing on our behalf sources obtained using Google Search as a tool.

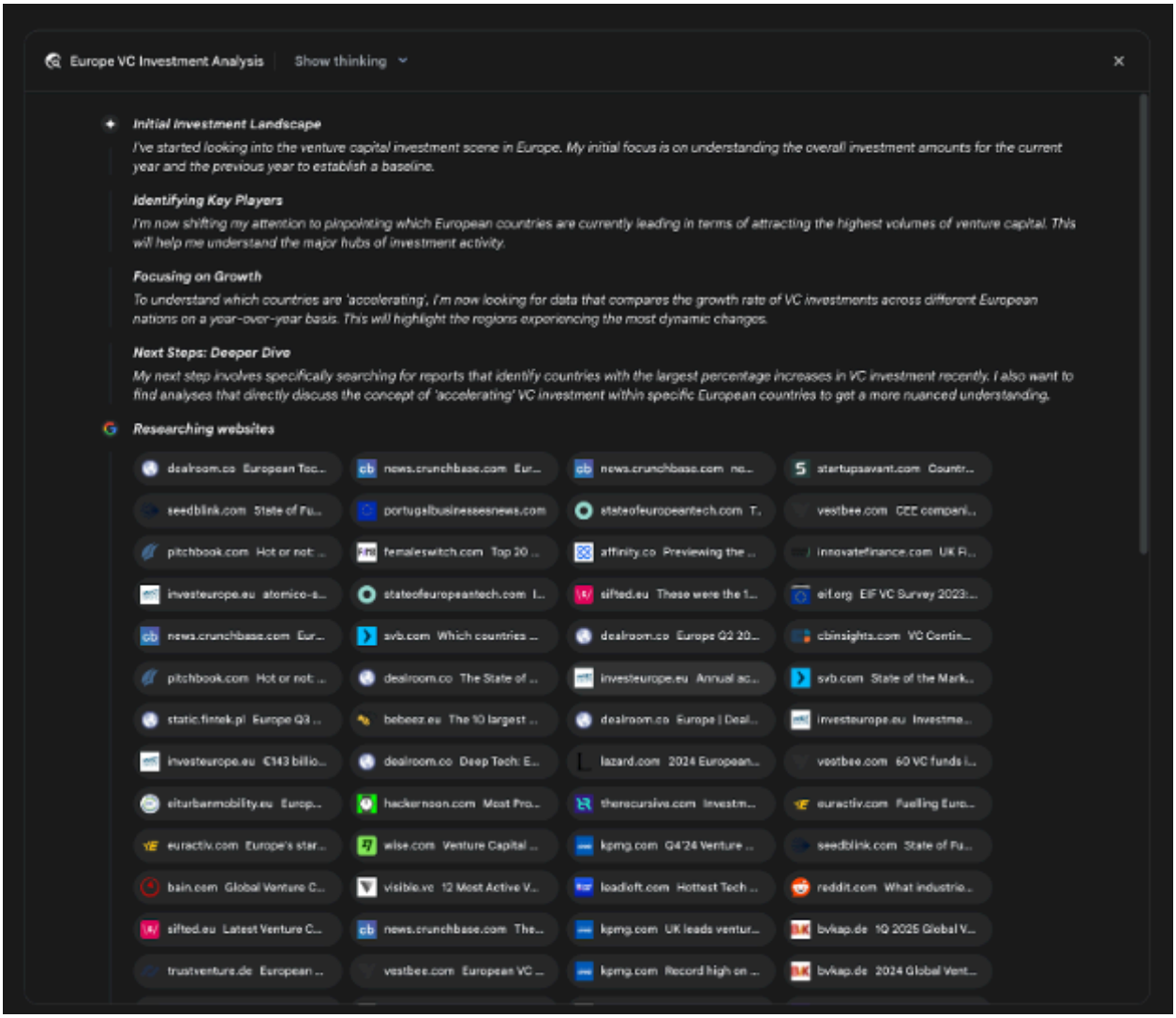

Figure 11: An example of Deep Research plan being executed, resulting in Google Search being used as a tool to search various web sources

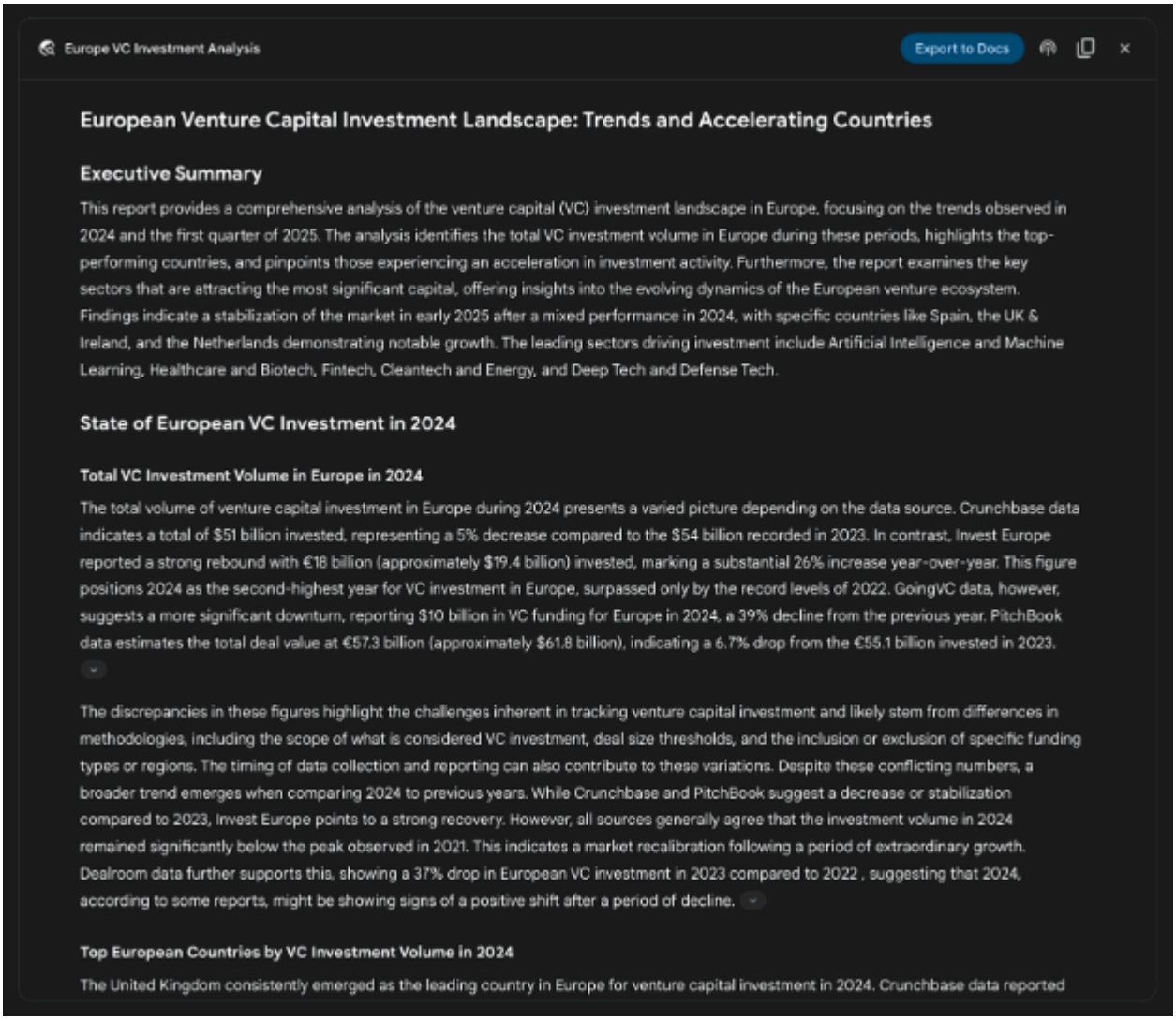

Figure 12: Final output generated by the Google Deep Research agent, analyzing on our behalf sources obtained using Google Search as a tool

圖 3：Google Deep Research 代理人產生的最終輸出，分析經由 Google Search 工具取得的來源。

Similarly, in academic exploration, the system serves as a powerful tool for conducting extensive literature reviews. It can identify and summarize foundational papers, trace the development of concepts across numerous publications, and map out emerging research fronts within a specific field, thereby accelerating the initial and most time–consuming phase of academic inquiry.

同樣地，在學術探索中，系統可作為進行大量文獻回顧的強大工具：它能辨識並摘要基礎論文、追蹤概念在多篇出版物中的發展，並描繪特定領域的新興研究方向，從而加速學術研究最初且最耗時的階段。

The efficiency of this approach stems from the automation of the iterative search–and–filter cycle, which is a core bottleneck in manual research. Comprehensiveness is achieved by the system's capacity to process a larger volume and variety of information sources than is typically feasible for a human researcher within a comparable timeframe. This broader scope of analysis helps to reduce the potential for selection bias and increases the likelihood of uncovering less obvious but potentially critical information, leading to a more robust and well–supported understanding of the subject matter.

此方法之所以高效，是因為它自動化了反覆的搜尋與篩選循環，而這正是人工研究的核心瓶頸。其全面性則來自於系統能在相近時間內處理比人類研究者更多元且更大量的資訊來源。更廣的分析範圍可降低選擇偏誤的風險，並提高找出不那麼顯眼但可能關鍵資訊的機會，帶來更穩健且更有根據的理解。

OpenAI Deep Research API

OpenAI Deep Research API

The OpenAI Deep Research API is a specialized tool designed to automate complex research tasks. It utilizes an advanced, agentic model that can independently reason, plan, and synthesize information from real–world sources. Unlike a simple Q&A model, it takes a high–level query and autonomously breaks it down into sub–questions, performs web searches using its built–in tools, and delivers a structured, citation–rich final report. The API provides direct programmatic access to this entire

process, using at the time of writing models like o3-deep-research-2025-06-26 for high-quality synthesis and the faster o4-mini-deep-research-2025-06-26 for latency-sensitive application

OpenAI Deep Research API 是用於自動化複雜研究任務的專門工具。它使用進階代理式模型，能獨立推理、規劃並從真實世界來源整合資訊。不同於簡單的 Q&A 模型，它會將高階問題拆成子問題，透過內建工具執行網路搜尋，並輸出具引用的結構化最終報告。該 API 提供整個流程的程式化存取，撰寫時常用 o3-deep-research-2025-06-26 進行高品質整合，也可用更快速的 o4-mini-deep-research-2025-06-26 用於延遲敏感應用。

The Deep Research API is useful because it automates what would otherwise be hours of manual research, delivering professional-grade, data-driven reports suitable for informing business strategy, investment decisions, or policy recommendations. Its key benefits include:

Deep Research API 的價值在於自動化原本需花費數小時的人工作業，產出可用於商業策略、投資決策或政策建議的專業級資料驅動報告。其主要優點包括：

- **Structured, Cited Output:** It produces well-organized reports with inline citations linked to source metadata, ensuring claims are verifiable and data-backed.

- **Transparency:** Unlike the abstracted process in ChatGPT, the API exposes all intermediate steps, including the agent's reasoning, the specific web search queries it executed, and any code it ran. This allows for detailed debugging, analysis, and a deeper understanding of how the final answer was constructed.

- **Extensibility:** It supports the Model Context Protocol (MCP), enabling developers to connect the agent to private knowledge bases and internal data sources, blending public web research with proprietary information.
- **結構化、具引用的輸出：**產出組織良好的報告，附帶與來源中繼資料連結的內嵌引用，確保主張可驗證且有資料支撐。

- **透明性：**不像 ChatGPT 的抽象流程，API 會揭示所有中間步驟，包括代理人的推理、執行的搜尋查詢與執行過的程式碼，使除錯與分析更細緻，並可深入理解最終答案如何形成。

- **可擴展性：**支援 Model Context Protocol（MCP），使開發者可把代理人連結到私有知識庫與內部資料來源，將公開網路研究與專有資訊結合。

To use the API, you send a request to the client.responses.create endpoint, specifying a model, an input prompt, and the tools the agent can use. The input typically includes a `system_message` that defines the agent's persona and desired output format, along with the `user_query`. You must also include the `web_search_preview` tool and can optionally add others like `code_interpreter` or custom MCP tools (see Chapter 10) for internal data.

要使用此 API，可對 client.responses.create 端點送出請求，指定模型、輸入提示與可用工具。輸入通常包含定義代理人角色與輸出格式的 `system_message`，以及 `user_query`。必須包含 `web_search_preview` 工具，也可選擇加入 `code_interpreter` 或自訂 MCP 工具（見第 10 章）以存取內部資料。

```
from openai import OpenAI


# Initialize the client with your API key
client = OpenAI(api_key="YOUR_OPENAI_API_KEY")


# Define the agent's role and the user's research question
system_message = """
You are a professional researcher preparing a structured, data-driven report.
Focus on data-rich insights, use reliable sources, and include inline citations.
"""

user_query = "Research the economic impact of semaglutide on global healthcare systems."


# Create the Deep Research API call
response = client.responses.create(
    model="o3-deep-research-2025-06-26",
    input=[
```

```
        {
            "role": "developer",
            "content": [{"type": "input_text", "text": system_message}],
        },
        {
            "role": "user",
            "content": [{"type": "input_text", "text": user_query}],
        },
    ],
    reasoning={"summary": "auto"},
    tools=[{"type": "web_search_preview"}],
)


# Access and print the final report from the response
final_report = response.output[-1].content[0].text
print(final_report)


# --- ACCESS INLINE CITATIONS AND METADATA ---
print("--- CITATIONS ---")
annotations = response.output[-1].content[0].annotations

if not annotations:
    print("No annotations found in the report.")
else:
    for i, citation in enumerate(annotations):
        # The text span the citation refers to
        cited_text = final_report[citation.start_index : citation.end_index]
        print(f"Citation {i + 1}:")
        print(f"  Cited Text: {cited_text}")
        print(f"  Title: {citation.title}")
        print(f"  URL: {citation.url}")
        print(f"  Location: chars {citation.start_index}—{citation.end_index}")
```

```python
print("\n" + "=" * 50 + "\n")


# --- INSPECT INTERMEDIATE STEPS ---
print("--- INTERMEDIATE STEPS ---")

# 1. Reasoning Steps: Internal plans and summaries generated by the model.
try:
    reasoning_step = next(item for item in response.output if item.type == "reasoning")
    print("\n[Found a Reasoning Step]")
    for summary_part in reasoning_step.summary:
        print(f"  - {summary_part.text}")
except StopIteration:
    print("\nNo reasoning steps found.")

# 2. Web Search Calls: The exact search queries the agent executed.
try:
    search_step = next(item for item in response.output if item.type == "web_search_call
    print("\n[Found a Web Search Call]")
    print(f"  Query Executed: '{search_step.action['query']}'")
    print(f"  Status: {search_step.status}")
except StopIteration:
    print("\nNo web search steps found.")

# 3. Code Execution: Any code run by the agent using the code interpreter.
try:
    code_step = next(item for item in response.output if item.type == "code_interpreter_
    print("\n[Found a Code Execution Step]")
    print("  Code Input:")
    print(f"  ```python\n{code_step.input}\n  ```")
    print("  Code Output:")
    print(f"  {code_step.output}")
except StopIteration:
```

```
    print("\nNo code execution steps found.")
```

This code snippet utilizes the OpenAI API to perform a “Deep Research”task. It starts by initializing the OpenAI client with your API key, which is crucial for authentication. Then, it defines the role of the AI agent as a professional researcher and sets the user’s research question about the economic impact of semaglutide. The code constructs an API call to the o3-deep-research-2025-06-26 model, providing the defined system message and user query as input. It also requests an automatic summary of the reasoning and enables web search capabilities. After making the API call, it extracts and prints the final generated report.

Subsequently, it attempts to access and display inline citations and metadata from the report’s annotations, including the cited text, title, URL, and location within the report. Finally, it inspects and prints details about the intermediate steps the model took, such as reasoning steps, web search calls (including the query executed), and any code execution steps if a code interpreter was used.

此程式碼片段使用 OpenAI API 執行「深度研究」任務。它先以 API 金鑰初始化 OpenAI client 以完成驗證，接著設定 AI 代理人角色為專業研究員，並指定使用者研究問題 (semaglutide 對全球醫療體系的經濟影響)。程式建立對 o3-deep-research-2025-06-26 模型的呼叫，輸入 system message 與 user query，並要求推理摘要與網路搜尋能力。完成呼叫後，取出並印出最終報告。

接著它嘗試讀取報告中的引用與中繼資料，包括被引用的文字、標題、URL 與所在位置。最後檢視模型的中間步驟，例如推理步驟、網路搜尋呼叫（含查詢內容）以及是否使用 code interpreter 的程式碼執行步驟。

At a Glance

一覽

What: Complex problems often cannot be solved with a single action and require foresight to achieve a desired outcome. Without a structured approach, an agentic system struggles to handle multifaceted requests that involve multiple steps and dependencies. This makes it difficult to break down high-level objectives into a manageable series of smaller, executable tasks. Consequently, the system fails to strategize effectively, leading to incomplete or incorrect results when faced with in-

tricate goals.

是什麼：複雜問題通常無法靠單一動作解決，需要前瞻性的多步驟規劃。缺乏結構化方法時，代理式系統難以處理包含多步驟與相依性的多面向需求，難以把高階目標拆解為可管理的可執行任務，導致策略不足、結果不完整或錯誤。

Why: The Planning pattern offers a standardized solution by having an agentic system first create a coherent plan to address a goal. It involves decomposing a high-level objective into a sequence of smaller, actionable steps or sub-goals. This allows the system to manage complex workflows, orchestrate various tools, and handle dependencies in a logical order. LLMs are particularly well-suited for this, as they can generate plausible and effective plans based on their vast training data. This structured approach transforms a simple reactive agent into a strategic executor that can proactively work towards a complex objective and even adapt its plan if necessary.

為什麼：規劃模式透過先建立一致的計畫來處理目標，提供標準化解法。它把高階目標拆解為一連串可執行的步驟或子目標，使系統能管理複雜流程、協調多種工具，並以邏輯順序處理相依關係。LLM 特別適合此任務，能依訓練資料生成合理且有效的計畫。這種結構化方法讓反應式代理人轉化為具策略性的執行者，能主動朝複雜目標前進並在必要時調整計畫。

Rule of thumb: Use this pattern when a user's request is too complex to be handled by a single action or tool. It is ideal for automating multi-step processes, such as generating a detailed research report, onboarding a new employee, or executing a competitive analysis. Apply the Planning pattern whenever a task requires a sequence of interdependent operations to reach a final, synthesized outcome.

經驗法則：當使用者需求過於複雜，無法以單一動作或工具完成時使用此模式。它適合自動化多步驟流程，例如產出詳細研究報告、員工入職流程或競品分析。凡是需要一連串相依操作才能達到最終整合結果的任務，都應採用規劃模式。

Visual summary

視覺摘要

Fig.4; Planning design pattern

圖 4：規劃設計模式

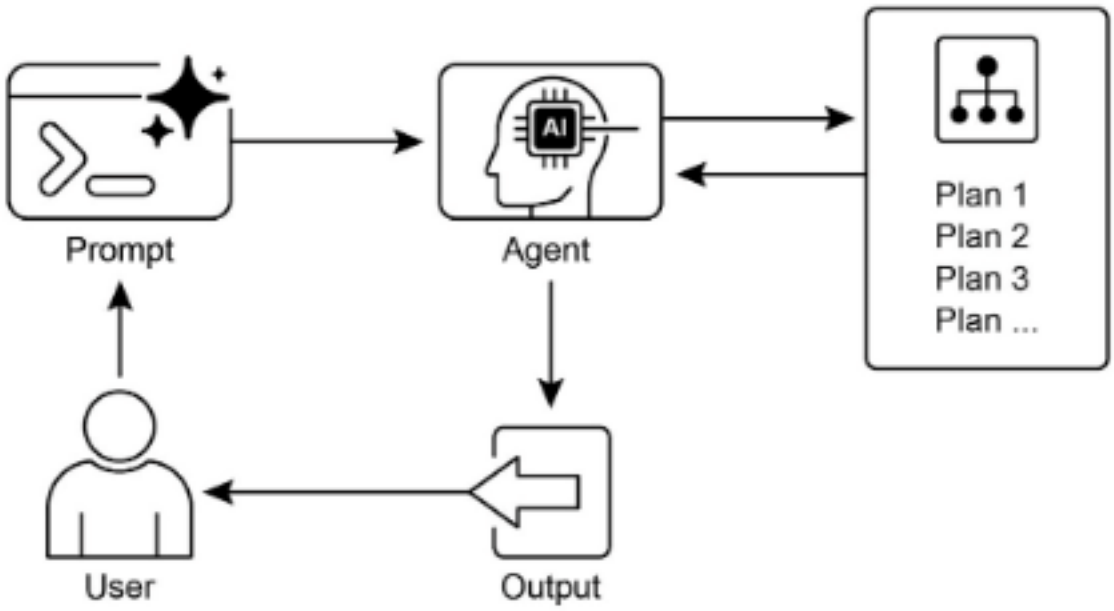

Figure 13: Planning Design Pattern

Key Takeaways

重點整理

- Planning enables agents to break down complex goals into actionable, sequential steps.

- It is essential for handling multi-step tasks, workflow automation, and navigating complex environments.

- LLMs can perform planning by generating step-by-step approaches based on task descriptions.

- Explicitly prompting or designing tasks to require planning steps encourages this behavior in agent frameworks.

- Google Deep Research is an agent analyzing on our behalf sources obtained

using Google Search as a tool. It reflects, plans, and executes

- 規劃讓代理人能將複雜目標拆解為可執行的序列步驟。
- 對多步任務、工作流程自動化與複雜環境導航至關重要。
- LLM 可依任務描述產生逐步計畫。
- 明確提示或設計需規劃步驟的任務，可促進代理框架中的規劃行為。
- Google Deep Research 是代表我們分析使用 Google Search 工具取得來源的代理人，能反思、規劃並執行。

Conclusion

結論

In conclusion, the Planning pattern is a foundational component that elevates agentic systems from simple reactive responders to strategic, goal–oriented executors. Modern large language models provide the core capability for this, autonomously decomposing high–level objectives into coherent, actionable steps. This pattern scales from straightforward, sequential task execution, as demonstrated by the CrewAI agent creating and following a writing plan, to more complex and dynamic systems. The Google DeepResearch agent exemplifies this advanced application, creating iterative research plans that adapt and evolve based on continuous information gathering. Ultimately, planning provides the essential bridge between human intent and automated execution for complex problems. By structuring a problem–solving approach, this pattern enables agents to manage intricate workflows and deliver comprehensive, synthesized results.

總結而言，規劃模式是將代理式系統從單純反應者提升為具策略、目標導向執行者的基礎元件。現代 LLM 提供核心能力，能自主將高階目標拆解為一致、可執行的步驟。此模式可從 CrewAI 代理人所示的簡單序列寫作計畫延伸到更複雜動態的系統。Google DeepResearch 代理人則展示進階應用，會隨持續資訊蒐集調整與演化研究計畫。最終，

規劃成為人類意圖與自動化執行之間的關鍵橋樑，透過結構化問題解決流程，使代理人能管理複雜工作流並產出完整綜合的成果。

References

參考資料

1. Google DeepResearch (Gemini Feature): gemini.google.com
2. OpenAI ,Introducing deep research https://openai.com/index/introducing-deep-research/
3. Perplexity, Introducing Perplexity Deep Research, https://www.perplexity.ai/hub/blog/introducing-perplexity-deep-research

---

Chapter 7: Multi-Agent Collaboration

第 7 章：多代理人協作

While a monolithic agent architecture can be effective for well-defined problems, its capabilities are often constrained when faced with complex, multi-domain tasks. The Multi-Agent Collaboration pattern addresses these limitations by structuring a system as a cooperative ensemble of distinct, specialized agents. This approach is predicated on the principle of task decomposition, where a high-level objective is broken down into discrete sub-problems. Each sub-problem is then assigned to an agent possessing the specific tools, data access, or reasoning capabilities best suited for that task.

雖然單體式代理人架構對明確定義的問題可能有效，但面對跨多領域的複雜任務時能力常受限制。多代理人協作模式透過把系統結構化為一組協同合作、具專長的代理人來解決這些限制。此方法建立在任務拆解原則上：將高階目標拆成離散子問題，再分配給具備最適工具、資料存取或推理能力的代理人。

For example, a complex research query might be decomposed and assigned to a Research Agent for information retrieval, a Data Analysis Agent for statistical pro-

cessing, and a Synthesis Agent for generating the final report. The efficacy of such a system is not merely due to the division of labor but is critically dependent on the mechanisms for inter-agent communication. This requires a standardized communication protocol and a shared ontology, allowing agents to exchange data, delegate sub-tasks, and coordinate their actions to ensure the final output is coherent.

例如，複雜研究問題可拆解並分配給資訊檢索的研究代理人、統計處理的資料分析代理人，以及產出最終報告的綜合代理人。此系統的成效不僅來自分工，更關鍵在於代理人間的溝通機制。這需要標準化的溝通協定與共享語彙體系，讓代理人能交換資料、委派子任務並協調行動，以確保最終輸出一致且連貫。

This distributed architecture offers several advantages, including enhanced modularity, scalability, and robustness, as the failure of a single agent does not necessarily cause a total system failure. The collaboration allows for a synergistic outcome where the collective performance of the multi-agent system surpasses the potential capabilities of any single agent within the ensemble.

這種分散式架構帶來多項優勢，包括更高的模組化、可擴展性與韌性，因為單一代理人的失敗不一定導致整體系統失效。協作能形成綜效，使多代理人系統的整體表現超越任何單一代理人的能力上限。

Multi-Agent Collaboration Pattern Overview

多代理人協作模式概覽

The Multi-Agent Collaboration pattern involves designing systems where multiple independent or semi-independent agents work together to achieve a common goal. Each agent typically has a defined role, specific goals aligned with the overall objective, and potentially access to different tools or knowledge bases. The power of this pattern lies in the interaction and synergy between these agents.

多代理人協作模式的核心是設計讓多個獨立或半獨立代理人共同完成目標的系統。每個代理人通常有明確角色、與整體目標一致的子目標，並可能擁有不同工具或知識庫。此模式的力量來自代理人之間的互動與協同。

Collaboration can take various forms:

協作可以有多種形式：

- **Sequential Handoffs:** One agent completes a task and passes its output to another agent for the next step in a pipeline (similar to the Planning pattern, but explicitly involving different agents).

- **Parallel Processing:** Multiple agents work on different parts of a problem simultaneously, and their results are later combined.

- **Debate and Consensus:** Multi-Agent Collaboration where Agents with varied perspectives and information sources engage in discussions to evaluate options, ultimately reaching a consensus or a more informed decision.

- **Hierarchical Structures:** A manager agent might delegate tasks to worker agents dynamically based on their tool access or plugin capabilities and synthesize their results. Each agent can also handle relevant groups of tools, rather than a single agent handling all the tools.

- **Expert Teams:** Agents with specialized knowledge in different domains (e.g., a researcher, a writer, an editor) collaborate to produce a complex output.
- **Sequential Handoffs（序列交接）：**一個代理人完成任務後將輸出交給下一個代理人處理後續步驟（類似規劃模式，但明確涉及不同代理人）。

- **Parallel Processing（平行處理）：**多個代理人同時處理問題的不同部分，之後再整合結果。

- **Debate and Consensus（辯論與共識）：**具不同觀點與資訊來源的代理人進行討論以評估選項，最終達成共識或更有資訊基礎的決策。

- **Hierarchical Structures（階層式結構）：**管理者代理人可依工具/外掛能力動態委派任務給工作代理人並綜合結果。每位代理人也可負責一組相關工具，而非由單一代理人處理所有工具。

- **Expert Teams（專家團隊）：**具不同領域專長的代理人（如研究者、寫作者、編輯）協作產出複雜成果。
- **Critic–Reviewer:** Agents create initial outputs such as plans, drafts, or answers. A second group of agents then critically assesses this output for adherence to policies, security, compliance, correctness, quality, and alignment with organizational objectives. The original creator or a final agent revises the output based on this feedback. This pattern is particularly effective for code generation, research writing, logic checking, and ensuring ethical alignment. The advantages of this approach include increased robustness, improved quality, and a reduced likelihood of hallucinations or errors.
- **Critic–Reviewer（評論者/審查者）：**代理人先產出初稿（如計畫、草稿或答案），第二組代理人再從政策、安全、合規、正確性、品質與組織目標一致性等面向進行嚴格評估。原作者或最終代理人依回饋修訂。此模式特別適用於程式碼生成、研究寫作、邏輯檢查與倫理一致性，優點是提升穩健性、品質並降低幻覺或錯誤。

A multi–agent system (see Fig.1) fundamentally comprises the delineation of agent roles and responsibilities, the establishment of communication channels through which agents exchange information, and the formulation of a task flow or interaction protocol that directs their collaborative endeavors.

多代理人系統（見圖 1）的基本要素是：清楚界定代理人角色與責任、建立資訊交換的溝通通道，以及制定任務流程或互動協定以引導協作。

Fig.1: Example of multi–agent system

圖 1：多代理人系統示例

Frameworks such as Crew AI and Google ADK are engineered to facilitate this paradigm by providing structures for the specification of agents, tasks, and their interactive procedures. This approach is particularly effective for challenges necessitating a variety of specialized knowledge, encompassing multiple discrete phases, or leveraging the advantages of concurrent processing and the corroboration of information across agents.

Crew AI 與 Google ADK 等框架為此範式提供代理人、任務與互動流程的規格化結構。這種方法特別適合需要多種專業知識、涵蓋多個離散階段，或需利用平行處理與代理人間資訊互證優勢的挑戰。

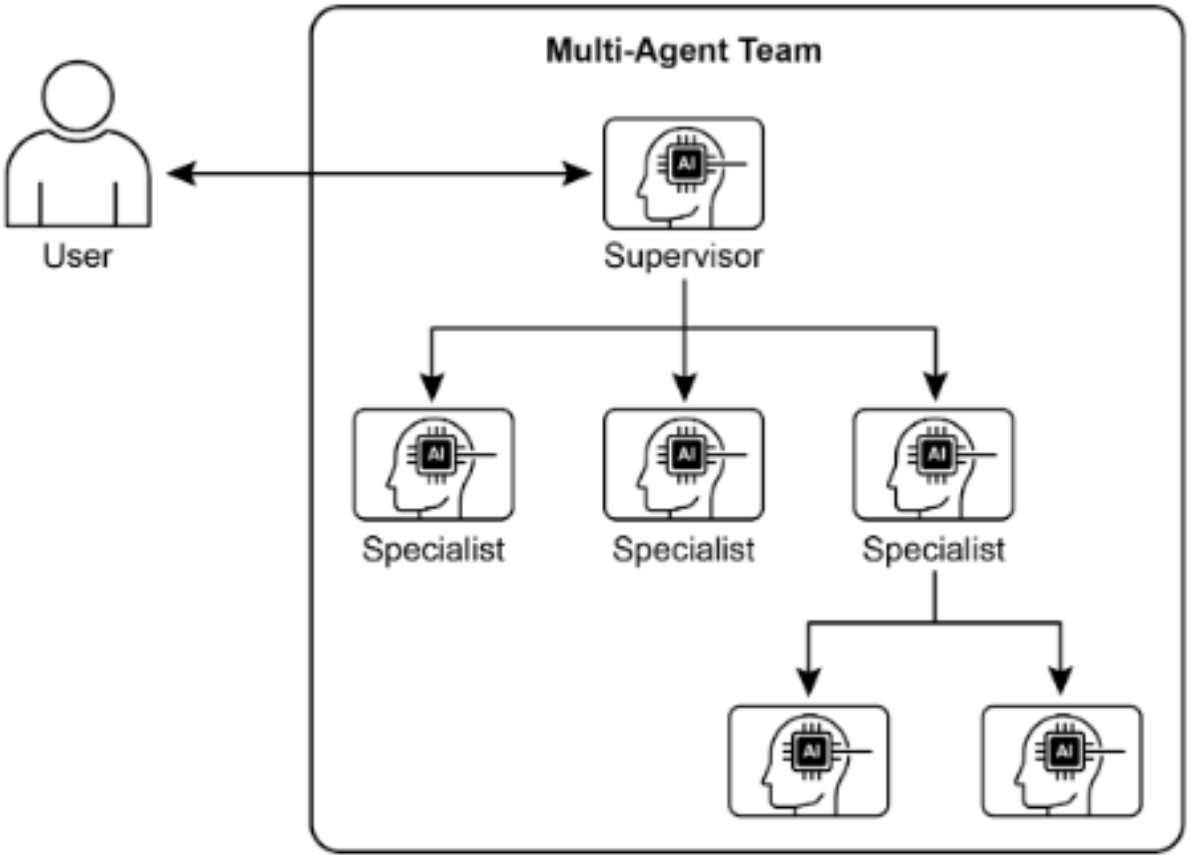

Figure 14: Multi–Agent System

Practical Applications & Use Cases

實務應用與使用情境

Multi-Agent Collaboration is a powerful pattern applicable across numerous domains:

多代理人協作是跨多領域皆適用的強大模式：

- **Complex Research and Analysis:** A team of agents could collaborate on a research project. One agent might specialize in searching academic databases, another in summarizing findings, a third in identifying trends, and a fourth in synthesizing the information into a report. This mirrors how a human research team might operate.

- **Software Development:** Imagine agents collaborating on building software. One agent could be a requirements analyst, another a code generator, a third a tester, and a fourth a documentation writer. They could pass outputs between each other to build and verify components.

- **Creative Content Generation:** Creating a marketing campaign could involve a market research agent, a copywriter agent, a graphic design agent (using image generation tools), and a social media scheduling agent, all working together.

- **Financial Analysis:** A multi-agent system could analyze financial markets. Agents might specialize in fetching stock data, analyzing news sentiment, performing technical analysis, and generating investment recommendations.

- **Customer Support Escalation:** A front-line support agent could handle initial queries, escalating complex issues to a specialist agent (e.g., a technical expert or a billing specialist) when needed, demonstrating a sequential handoff based on problem complexity.

- **Supply Chain Optimization:** Agents could represent different nodes in a supply chain (suppliers, manufacturers, distributors) and collaborate to optimize

inventory levels, logistics, and scheduling in response to changing demand or disruptions.

- **Network Analysis & Remediation**: Autonomous operations benefit greatly from an agentic architecture, particularly in failure pinpointing. Multiple agents can collaborate to triage and remediate issues, suggesting optimal actions. These agents can also integrate with traditional machine learning models and tooling, leveraging existing systems while simultaneously offering the advantages of Generative AI.
- **複雜研究與分析：**代理人團隊協作完成研究專案，例如一個專注學術資料庫搜尋、另一個負責摘要、第三個辨識趨勢、第四個綜合成報告，類似人類研究團隊的運作方式。
- **軟體開發：**想像代理人共同開發軟體：需求分析、程式碼生成、測試與文件撰寫分工協作，彼此交接輸出以建構與驗證元件。
- **創意內容生成：**行銷活動可由市場研究代理人、文案代理人、平面設計代理人（使用影像生成工具）與社群排程代理人共同完成。
- **財務分析：**多代理人系統可分析金融市場，代理人分工取得股價、分析新聞情緒、做技術分析並產生投資建議。
- **客服升級：**一線客服代理人處理初步問題，遇到複雜議題時升級給專家代理人（如技術或帳務專家），展現依問題複雜度進行序列交接。
- **供應鏈最佳化：**代理人可代表供應鏈不同節點（供應商、製造商、經銷商）協作，因應需求變化或干擾最佳化庫存、物流與排程。
- **網路分析與修復：**自主營運架構在故障定位上受益甚多，多代理人可協作分診與修復，提出最佳行動建議，也能整合傳統 ML 模型與工具，兼具既有系統與生成式 AI 的優勢。

The capacity to delineate specialized agents and meticulously orchestrate their interrelationships empowers developers to construct systems exhibiting enhanced modularity, scalability, and the ability to address complexities that would prove insurmountable for a singular, integrated agent.

透過明確分工並精細編排代理人間關係，開發者能建構具更高模組化、可擴展性、且能處理單一整合代理人難以應付之複雜性的系統。

Multi-Agent Collaboration: Exploring Interrelationships and Communication Structures

多代理人協作：關係與溝通結構

Understanding the intricate ways in which agents interact and communicate is fundamental to designing effective multi-agent systems. As depicted in Fig. 2, a spectrum of interrelationship and communication models exists, ranging from the simplest single-agent scenario to complex, custom-designed collaborative frameworks. Each model presents unique advantages and challenges, influencing the overall efficiency, robustness, and adaptability of the multi-agent system.

理解代理人間互動與溝通的細微方式，是設計有效多代理人系統的基礎。如圖 2 所示，存在從最簡單的單代理人情境到複雜客製協作架構的多種關係與溝通模型。每種模型都有其優勢與挑戰，影響多代理人系統的效率、韌性與適應性。

1. Single Agent

1. 單一代理人

At the most basic level, a "Single Agent" operates autonomously without direct interaction or communication with other entities. While this model is straightforward to implement and manage, its capabilities are inherently limited by the individual agent's scope and resources. It is suitable for tasks that are decomposable into independent sub-problems, each solvable by a single, self-sufficient agent.

在最基本層級，「單一代理人」自主運作，沒有與其他實體的直接互動或溝通。此模型易於實作與管理，但能力受限於單一代理人的範圍與資源。適合可拆成獨立子問題、且每個子問題由單一自足代理人可解的任務。

2. Network

2. 網路型

The “Network”model represents a significant step towards collaboration, where multiple agents interact directly with each other in a decentralized fashion. Communication typically occurs peer-to-peer, allowing for the sharing of information, resources, and even tasks. This model fosters resilience, as the failure of one agent does not necessarily cripple the entire system. However, managing communication overhead and ensuring coherent decision-making in a large, unstructured network can be challenging.

「網路型」模型代表合作的一大步，多個代理人以去中心化方式直接互動，通常採對等(peer-to-peer）溝通，共享資訊、資源甚至任務。此模型具韌性，因單一代理人失效不一定癱瘓全系統。但在大型、非結構化網路中管理溝通負擔並確保決策一致性會較具挑戰。

3. Supervisor

3. 監督者

In the “Supervisor”model, a dedicated agent, the “supervisor,”oversees and coordinates the activities of a group of subordinate agents. The supervisor acts as a central hub for communication, task allocation, and conflict resolution. This hierarchical structure offers clear lines of authority and can simplify management and control. However, it introduces a single point of failure (the supervisor) and can become a bottleneck if the supervisor is overwhelmed by a large number of subordinates or complex tasks.

「監督者」模型由一個專責代理人（監督者）統籌與協調一群下屬代理人的活動。監督者作為溝通、任務分配與衝突解決的中心樞紐。此階層式結構權責清楚，便於管理與控制，但也引入單點失效風險；若監督者面對大量下屬或複雜任務，可能成為瓶頸。

4. Supervisor as a Tool

4. 監督者作為工具

This model is a nuanced extension of the “Supervisor”concept, where the supervisor’s role is less about direct command and control and more about providing resources, guidance, or analytical support to other agents. The supervisor might offer tools, data, or computational services that enable other agents to perform their tasks more effectively, without necessarily dictating their every action. This approach aims to leverage the supervisor’s capabilities without imposing rigid top–down control.

此模型是「監督者」概念的細緻延伸，監督者不再是直接指揮控制，而是提供資源、指引或分析支援。監督者可能提供工具、資料或計算服務，讓其他代理人更有效執行任務，而不必事事指揮。此方法旨在利用監督者能力，同時避免僵化的自上而下控制。

5. Hierarchical

5. 階層式

The “Hierarchical”model expands upon the supervisor concept to create a multi–layered organizational structure. This involves multiple levels of supervisors, with higher–level supervisors overseeing lower–level ones, and ultimately, a collection of operational agents at the lowest tier. This structure is well–suited for complex problems that can be decomposed into sub–problems, each managed by a specific layer of the hierarchy. It provides a structured approach to scalability and complexity management, allowing for distributed decision–making within defined boundaries.

「階層式」模型在監督者概念上擴展為多層組織結構，包含多層級監督者，上層管理下層，最底層為執行代理人。此結構適合可拆解為多個子問題的複雜任務，每層負責管理特定子問題。它提供結構化的擴展與複雜度管理，讓決策在明確邊界內分散進行。

Fig. 2: Agents communicate and interact in various ways.

圖 2：代理人以不同方式溝通與互動。

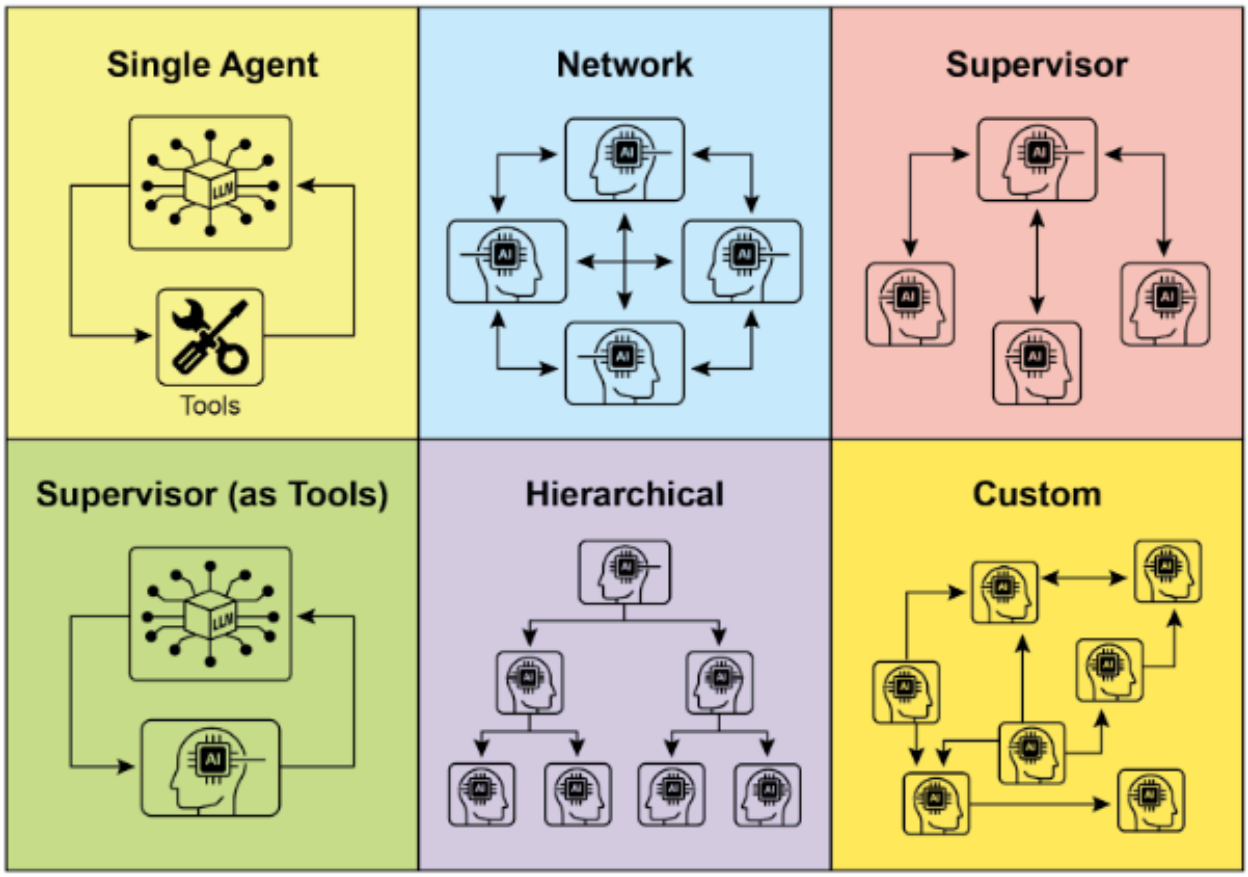

Figure 15: Agents Communicate and Interact in Various Ways

6. Custom

6. 客製型

The “Custom”model represents the ultimate flexibility in multi–agent system design. It allows for the creation of unique interrelationship and communication structures tailored precisely to the specific requirements of a given problem or application. This can involve hybrid approaches that combine elements from the previously mentioned models, or entirely novel designs that emerge from the unique constraints and opportunities of the environment. Custom models often arise from the need to optimize for specific performance metrics, handle highly dynamic environments, or incorporate domain–specific knowledge into the system’s architecture. Designing and implementing custom models typically requires a deep understanding of multi–agent systems principles and careful consideration of communication protocols, coordination mechanisms, and emergent behaviors.

「客製型」模型代表多代理人系統設計的最高彈性，允許建立完全依特定問題/應用需求量身打造的關係與溝通結構。這可包含混合先前模型元素的混合式設計，或因環境獨特限制與機會而產生的全新設計。客製模型常因需優化特定效能指標、處理高度動態環境或把領域知識納入架構而出現。設計與實作客製模型通常需要對多代理人原理有深入理解，並審慎考量溝通協定、協調機制與湧現行為。

In summary, the choice of interrelationship and communication model for a multi–agent system is a critical design decision. Each model offers distinct advantages and disadvantages, and the optimal choice depends on factors such as the complexity of the task, the number of agents, the desired level of autonomy, the need for robustness, and the acceptable communication overhead. Future advancements in multi–agent systems will likely continue to explore and refine these models, as well as develop new paradigms for collaborative intelligence.

總結而言，選擇多代理人系統的關係與溝通模型是一項關鍵設計決策。每種模型都有不同的優缺點，最佳選擇取決於任務複雜度、代理人數量、所需自主性、韌性需求與可接受的溝通負擔。多代理人系統的未來發展將持續探索與精煉這些模型，並發展新的協作智慧範式。

Hands-On code (Crew AI)

動手實作（Crew AI）

This Python code defines an AI-powered crew using the CrewAI framework to generate a blog post about AI trends. It starts by setting up the environment, loading API keys from a .env file. The core of the application involves defining two agents: a researcher to find and summarize AI trends, and a writer to create a blog post based on the research.

此 Python 程式碼使用 CrewAI 框架建立 AI 團隊，用來生成關於 AI 趨勢的部落格文章。它先設定環境並從.env 檔載入 API 金鑰。核心流程是定義兩個代理人：研究者負責尋找並摘要 AI 趨勢，寫作者根據研究結果撰寫文章。

Two tasks are defined accordingly: one for researching the trends and another for writing the blog post, with the writing task depending on the output of the research task. These agents and tasks are then assembled into a Crew, specifying a sequential process where tasks are executed in order. The Crew is initialized with the agents, tasks, and a language model (specifically the "gemini-2.0-flash"model). The main function executes this crew using the kickoff() method, orchestrating the collaboration between the agents to produce the desired output. Finally, the code prints the final result of the crew's execution, which is the generated blog post.

因此定義了兩個任務：一個研究趨勢、另一個寫作文章，且寫作任務依賴研究任務的輸出。接著將代理人與任務組裝成 Crew，指定序列流程依序執行。Crew 以代理人、任務與語言模型（"gemini-2.0-flash"）初始化。主函式用 kickoff() 執行 Crew，協調代理人合作產出結果，最後印出生成的文章。

```
import os
from dotenv import load_dotenv
from crewai import Agent, Task, Crew, Process
from langchain_google_genai import ChatGoogleGenerativeAI


def setup_environment():
    """Loads environment variables and checks for the required API key."""
    load_dotenv()
```

```
    if not os.getenv("GOOGLE_API_KEY"):
        raise ValueError("GOOGLE_API_KEY not found. Please set it in your .env file.")


def main():
    """
    Initializes and runs the AI crew for content creation using the latest Gemini mode
    """
    setup_environment()

    # Define the language model to use.
    # Updated to a model from the Gemini 2.0 series for better performance and feature
    # For cutting-edge (preview) capabilities, you could use "gemini-2.5-flash".
    llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")

    # Define Agents with specific roles and goals
    researcher = Agent(
        role='Senior Research Analyst',
        goal='Find and summarize the latest trends in AI.',
        backstory="You are an experienced research analyst with a knack for identifying
        verbose=True,
        allow_delegation=False,
    )

    writer = Agent(
        role='Technical Content Writer',
        goal='Write a clear and engaging blog post based on research findings.',
        backstory="You are a skilled writer who can translate complex technical topics i
        verbose=True,
        allow_delegation=False,
    )

    # Define Tasks for the agents
    research_task = Task(
```

```
        description="Research the top 3 emerging trends in Artificial Intelligence in 20
        expected_output="A detailed summary of the top 3 AI trends, including key points
        agent=researcher,
    )

    writing_task = Task(
        description="Write a 500-word blog post based on the research findings. The post
        expected_output="A complete 500-word blog post about the latest AI trends.",
        agent=writer,
        context=[research_task],
    )

    # Create the Crew
    blog_creation_crew = Crew(
        agents=[researcher, writer],
        tasks=[research_task, writing_task],
        process=Process.sequential,
        llm=llm,
        verbose=2,  # Set verbosity for detailed crew execution logs
    )

    # Execute the Crew
    print("## Running the blog creation crew with Gemini 2.0 Flash... ##")
    try:
        result = blog_creation_crew.kickoff()
        print("\n------------------\n")
        print("## Crew Final Output ##")
        print(result)
    except Exception as e:
        print(f"\nAn unexpected error occurred: {e}")


if __name__ == "__main__":
    main()
```

We will now delve into further examples within the Google ADK framework, with particular emphasis on hierarchical, parallel, and sequential coordination paradigms, alongside the implementation of an agent as an operational instrument.

接下來我們將進一步深入 Google ADK 框架中的範例，特別著重於階層式、平行式與序列式協調範式，以及將代理人作為操作性工具的實作。

Hands-on Code (Google ADK)

動手實作（Google ADK）

The following code example demonstrates the establishment of a hierarchical agent structure within the Google ADK through the creation of a parent-child relationship. The code defines two types of agents: LlmAgent and a custom TaskExecutor agent derived from BaseAgent. The TaskExecutor is designed for specific, non-LLM tasks and in this example, it simply yields a "Task finished successfully"event. An LlmAgent named greeter is initialized with a specified model and instruction to act as a friendly greeter. The custom TaskExecutor is instantiated as `task_doer`. A parent LlmAgent called coordinator is created, also with a model and instructions. The coordinator's instructions guide it to delegate greetings to the greeter and task execution to the `task_doer`. The greeter and `task_doer` are added as sub-agents to the coordinator, establishing a parent-child relationship. The code then asserts that this relationship is correctly set up. Finally, it prints a message indicating that the agent hierarchy has been successfully created.

以下程式碼示範如何在 Google ADK 中建立階層式代理人結構，透過父子關係來組織代理人。此程式定義兩種代理人：LlmAgent 與由 BaseAgent 衍生的自訂 TaskExecutor。TaskExecutor 用於非 LLM 的特定任務，本例中只是產生「Task finished successfully」事件。一個名為 greeter 的 LlmAgent 以指定模型與指令初始化，用於友善問候。自訂 TaskExecutor 則以 `task_doer` 實例化。再建立名為 coordinator 的父級 LlmAgent，並透過指令引導其把問候委派給 greeter、把任務委派給 `task_doer`。greeter 與 `task_doer` 被加入 coordinator 的子代理人，形成父子關係。程式碼接著確認關係設定正確，最後印出訊息表示階層建立成功。

```
from typing import AsyncGenerator
```

```python
from google.adk.agents import LlmAgent, BaseAgent
from google.adk.agents.invocation_context import InvocationContext
from google.adk.events import Event


# Correctly implement a custom agent by extending BaseAgent
class TaskExecutor(BaseAgent):
    """A specialized agent with custom, non-LLM behavior."""
    name: str = "TaskExecutor"
    description: str = "Executes a predefined task."

    async def _run_async_impl(self, context: InvocationContext) -> AsyncGenerator[Event,
        """Custom implementation logic for the task."""
        # This is where your custom logic would go.
        # For this example, we'll just yield a simple event.
        yield Event(author=self.name, content="Task finished successfully.")


# Define individual agents with proper initialization
# LlmAgent requires a model to be specified.
greeter = LlmAgent(
    name="Greeter",
    model="gemini-2.0-flash-exp",
    instruction="You are a friendly greeter.",
)

# Instantiate our concrete custom agent
task_doer = TaskExecutor()

# Create a parent agent and assign its sub-agents
# The parent agent's description and instructions should guide its delegation logic.
coordinator = LlmAgent(
    name="Coordinator",
    model="gemini-2.0-flash-exp",
```

```
    description="A coordinator that can greet users and execute tasks.",
    instruction="When asked to greet, delegate to the Greeter. When asked to perform a t
    sub_agents=[
        greeter,
        task_doer,
    ],
)

# The ADK framework automatically establishes the parent-child relationships.
# These assertions will pass if checked after initialization.
assert greeter.parent_agent == coordinator
assert task_doer.parent_agent == coordinator

print("Agent hierarchy created successfully.")
```

This code excerpt illustrates the employment of the LoopAgent within the Google ADK framework to establish iterative workflows. The code defines two agents: ConditionChecker and ProcessingStep. ConditionChecker is a custom agent that checks a "status"value in the session state. If the "status"is "completed", ConditionChecker escalates an event to stop the loop. Otherwise, it yields an event to continue the loop. `ProcessingStep` is an LlmAgent using the "gemini-2.0-flash-exp"model. Its instruction is to perform a task and set the session `status` to "completed"if it's the final step. A LoopAgent named StatusPoller is created. StatusPoller is configured with `max_iterations=10`. StatusPoller includes both ProcessingStep and an instance of ConditionChecker as sub-agents. The LoopAgent will execute the sub-agents sequentially for up to 10 iterations, stopping if ConditionChecker finds the status is "completed".

這段程式碼示範在 Google ADK 中使用 LoopAgent 建立迭代式流程。程式定義兩個代理人：ConditionChecker 與 ProcessingStep。ConditionChecker 是自訂代理人，檢查 session state 中的"status"。若為"completed"，則升級事件以停止迴圈；否則產生事件以繼續迴圈。`ProcessingStep` 是使用"gemini-2.0-flash-exp"模型的 LlmAgent，指令要求執行任務並在最後一步把 session 的 `status` 設為"completed"。建立名為 StatusPoller 的 LoopAgent，並設定 `max_iterations=10`，子代理人包含 ProcessingStep 與 ConditionChecker。LoopAgent 會依序執行子代理人最多 10 次，直到 ConditionChecker 判斷完成為止。

```python
import asyncio
from typing import AsyncGenerator

from google.adk.agents import LoopAgent, LlmAgent, BaseAgent
from google.adk.events import Event, EventActions
from google.adk.agents.invocation_context import InvocationContext


# Best Practice: Define custom agents as complete, self-describing classes.
class ConditionChecker(BaseAgent):
    """A custom agent that checks for a 'completed' status in the session state."""
    name: str = "ConditionChecker"
    description: str = "Checks if a process is complete and signals the loop to stop."

    async def _run_async_impl(
        self, context: InvocationContext
    ) -> AsyncGenerator[Event, None]:
        """Checks state and yields an event to either continue or stop the loop."""
        status = context.session.state.get("status", "pending")
        is_done = status == "completed"

        if is_done:
            # Escalate to terminate the loop when the condition is met.
            yield Event(author=self.name, actions=EventActions(escalate=True))
        else:
            # Yield a simple event to continue the loop.
            yield Event(author=self.name, content="Condition not met, continuing loop.")


# Correction: The LlmAgent must have a model and clear instructions.
process_step = LlmAgent(
    name="ProcessingStep",
    model="gemini-2.0-flash-exp",
    instruction=(
```

```
        "You are a step in a longer process. Perform your task. "
        "If you are the final step, update session state by setting 'status' to 'complet
    ),
)


# The LoopAgent orchestrates the workflow.
poller = LoopAgent(
    name="StatusPoller",
    max_iterations=10,
    sub_agents=[
        process_step,
        ConditionChecker(),  # Instantiating the well-defined custom agent.
    ],
)

# This poller will now execute 'process_step'
# and then 'ConditionChecker' repeatedly until the status is 'completed'
# or 10 iterations have passed.
```

This code excerpt elucidates the SequentialAgent pattern within the Google ADK, engineered for the construction of linear workflows. This code defines a sequential agent pipeline using the google.adk.agents library. The pipeline consists of two agents, step1 and step2. step1 is named `Step1_Fetch` and its output will be stored in the session state under the key `data`. step2 is named `Step2_Process` and is instructed to analyze the information stored in `session.state["data"]` and provide a summary. The SequentialAgent named "MyPipeline"orchestrates the execution of these sub-agents. When the pipeline is run with an initial input, step1 will execute first. The response from step1 will be saved into the session state under the key "data". Subsequently, step2 will execute, utilizing the information that step1 placed into the state as per its instruction. This structure allows for building workflows where the output of one agent becomes the input for the next. This is a common pattern in creating multi-step AI or data processing pipelines.

這段程式碼說明 Google ADK 的 SequentialAgent 模式，用於建構線性流程。程式使用 google.adk.agents 建立順序式代理管線，包含兩個代理人 step1 與 step2。step1 名為

Step1_Fetch，輸出會存入 session state 的 data。step2 名為 Step2_Process，指示它分析 session.state["data"] 的資訊並提供摘要。名為“MyPipeline”的 SequentialAgent 負責協調子代理人執行。流程啟動後，step1 先執行並把回應存到 data，接著 step2 再使用該資料做摘要。此結構讓一個代理人的輸出成為下一個代理人的輸入，是常見的多步驟 AI 或資料處理管線模式。

```
from google.adk.agents import SequentialAgent, Agent


# This agent's output will be saved to session.state["data"]
step1 = Agent(
    name="Step1_Fetch",
    output_key="data",
)

# This agent will use the data from the previous step.
# We instruct it on how to find and use this data.
step2 = Agent(
    name="Step2_Process",
    instruction="Analyze the information found in state['data'] and provide a summary.",
)

pipeline = SequentialAgent(
    name="MyPipeline",
    sub_agents=[step1, step2],
)

# When the pipeline is run with an initial input, Step1 will execute,
# its response will be stored in session.state["data"], and then
# Step2 will execute, using the information from the state as instructed.
```

The following code example illustrates the ParallelAgent pattern within the Google ADK, which facilitates the concurrent execution of multiple agent tasks. The data_gatherer is designed to run two sub-agents concurrently: weather_fetcher and news_fetcher. The weather_fetcher agent is instructed to get the weather for a given location and store the result in session.state["weather_data"]. Similarly,

the news_fetcher agent is instructed to retrieve the top news story for a given topic and store it in session.state["news_data"]. Each sub-agent is configured to use the "gemini-2.0-flash-exp"model. The ParallelAgent orchestrates the execution of these sub-agents, allowing them to work in parallel. The results from both weather_fetcher and news_fetcher would be gathered and stored in the session state. Finally, the example shows how to access the collected weather and news data from the final_state after the agent's execution is complete.

以下程式碼示範 Google ADK 的 ParallelAgent 模式，支援多個代理任務的併發執行。data_gatherer 設計為同時執行兩個子代理人：weather_fetcher 與 news_fetcher。weather_fetcher 被指示取得指定地點天氣並存入 session.state["weather_data"]；news_fetcher 被指示取得指定主題的頭條新聞並存入 session.state["news_data"]。每個子代理人使用"gemini-2.0-flash-exp"模型。ParallelAgent 會協調兩個子代理人平行運作，完成後把結果存入 session state。範例最後示範如何在執行完成後從 final_state 取得天氣與新聞資料。

```
from google.adk.agents import Agent, ParallelAgent


# It's better to define the fetching logic as tools for the agents.
# For simplicity in this example, we'll embed the logic in the agent's instruction.
# In a real-world scenario, you would use tools.

# Define the individual agents that will run in parallel
weather_fetcher = Agent(
    name="weather_fetcher",
    model="gemini-2.0-flash-exp",
    instruction="Fetch the weather for the given location and return only the weather re
    output_key="weather_data",  # The result will be stored in session.state["weather_d
)

news_fetcher = Agent(
    name="news_fetcher",
    model="gemini-2.0-flash-exp",
    instruction="Fetch the top news story for the given topic and return only that story
    output_key="news_data",  # The result will be stored in session.state["news_data"]
```

```
)

# Create the ParallelAgent to orchestrate the sub-agents
data_gatherer = ParallelAgent(
    name="data_gatherer",
    sub_agents=[
        weather_fetcher,
        news_fetcher,
    ],
)
```

The provided code segment exemplifies the "Agent as a Tool"paradigm within the Google ADK, enabling an agent to utilize the capabilities of another agent in a manner analogous to function invocation. Specifically, the code defines an image generation system using Google's LlmAgent and AgentTool classes. It consists of two agents: a parent `artist_agent` and a sub-agent `image_generator_agent`. The `generate_image` function is a simple tool that simulates image creation, returning mock image data. The `image_generator_agent` is responsible for using this tool based on a text prompt it receives. The `artist_agent`'s role is to first invent a creative image prompt. It then calls the `image_generator_agent` through an AgentTool wrapper. The AgentTool acts as a bridge, allowing one agent to use another agent as a tool. When the `artist_agent` calls the `image_tool`, the AgentTool invokes the `image_generator_agent` with the artist's invented prompt. The `image_generator_agent` then uses the `generate_image` function with that prompt. Finally, the generated image (or mock data) is returned back up through the agents. This architecture demonstrates a layered agent system where a higher-level agent orchestrates a lower-level, specialized agent to perform a task.

此程式片段示範 Google ADK 中的「代理人作為工具」範式，讓一個代理人以類似函式呼叫的方式使用另一個代理人的能力。程式使用 Google 的 LlmAgent 與 AgentTool 建立影像生成系統，由父代理人 `artist_agent` 與子代理人 `image_generator_agent` 組成。`generate_image` 函式作為工具模擬影像生成，回傳假資料。`image_generator_agent` 會根據收到的文字提示使用該工具。`artist_agent` 先創造一段創意影像提示，再透過 AgentTool 呼叫 `image_generator_agent`。AgentTool 是橋樑，使一個代理人能將另一個代理人當成工具使用。`artist_agent` 呼叫 `image_tool` 後，AgentTool 會用其創意提示呼叫 `image_generator_agent`，後者再使用 `generate_image` 生成影像並回傳。此架構展

示了分層代理系統：高階代理人負責協調，低階專門代理人負責執行。

```
from google.adk.agents import LlmAgent
from google.adk.tools import agent_tool
from google.genai import types


# 1. A simple function tool for the core capability.
# This follows the best practice of separating actions from reasoning.
def generate_image(prompt: str) -> dict:
    """
    Generates an image based on a textual prompt.

    Args:
        prompt: A detailed description of the image to generate.

    Returns:
        A dictionary with the status and the generated image bytes.
    """
    print(f"TOOL: Generating image for prompt: '{prompt}'")
    # In a real implementation, this would call an image generation API.
    # For this example, we return mock image data.
    mock_image_bytes = b"mock_image_data_for_a_cat_wearing_a_hat"
    return {
        "status": "success",
        # The tool returns the raw bytes, the agent will handle the Part creation.
        "image_bytes": mock_image_bytes,
        "mime_type": "image/png",
    }


# 2. Refactor the ImageGeneratorAgent into an LlmAgent.
# It now correctly uses the input passed to it.
image_generator_agent = LlmAgent(
    name="ImageGen",
```

```
    model="gemini-2.0-flash",
    description="Generates an image based on a detailed text prompt.",
    instruction=(
        "You are an image generation specialist. Your task is to take the user's request
        "and use the `generate_image` tool to create the image. "
        "The user's entire request should be used as the 'prompt' argument for the tool.
        "After the tool returns the image bytes, you MUST output the image."
    ),
    tools=[generate_image],
)


# 3. Wrap the corrected agent in an AgentTool.
# The description here is what the parent agent sees.
image_tool = agent_tool.AgentTool(
    agent=image_generator_agent,
    description="Use this tool to generate an image. The input should be a descriptive p
)


# 4. The parent agent remains unchanged. Its logic was correct.
artist_agent = LlmAgent(
    name="Artist",
    model="gemini-2.0-flash",
    instruction=(
        "You are a creative artist. First, invent a creative and descriptive prompt for
        "Then, use the `ImageGen` tool to generate the image using your prompt."
    ),
    tools=[image_tool],
)
```

At a Glance

一覽

What: Complex problems often exceed the capabilities of a single, monolithic LLM–based agent. A solitary agent may lack the diverse, specialized skills or access to the specific tools needed to address all parts of a multifaceted task. This limitation creates a bottleneck, reducing the system's overall effectiveness and scalability. As a result, tackling sophisticated, multi–domain objectives becomes inefficient and can lead to incomplete or suboptimal outcomes.

是什麼：複雜問題往往超出單一、單體式 LLM 代理人的能力。單一代理人可能缺乏多元、專門技能或無法存取處理多面向任務所需的特定工具。這會形成瓶頸，降低系統的整體效能與可擴展性，導致處理跨領域目標變得低效，甚至產生不完整或次佳結果。

Why: The Multi–Agent Collaboration pattern offers a standardized solution by cre–ating a system of multiple, cooperating agents. A complex problem is broken down into smaller, more manageable sub–problems. Each sub–problem is then assigned to a specialized agent with the precise tools and capabilities required to solve it. These agents work together through defined communication protocols and interac–tion models like sequential handoffs, parallel workstreams, or hierarchical delegation. This agentic, distributed approach creates a synergistic effect, allowing the group to achieve outcomes that would be impossible for any single agent.

為什麼：多代理人協作模式透過建立多個協同代理人的系統提供標準化解法。複雜問題被拆成較小、可管理的子問題，再分配給具備精準工具與能力的專門代理人。這些代理人透過定義好的溝通協定與互動模型（如序列交接、平行工作流或階層委派）協作，形成分散式的綜效，使團隊達到單一代理人無法達成的成果。

Rule of thumb: Use this pattern when a task is too complex for a single agent and can be decomposed into distinct sub–tasks requiring specialized skills or tools. It is ideal for problems that benefit from diverse expertise, parallel processing, or a structured workflow with multiple stages, such as complex research and analysis, software development, or creative content generation.

經驗法則：當任務對單一代理人過於複雜，且可拆解為需要專門技能或工具的不同子任務時，採用此模式最合適。它適用於需要多元專業、平行處理或多階段結構化流程的問題，例如複雜研究分析、軟體開發或創意內容生成。

Visual summary:

視覺摘要：

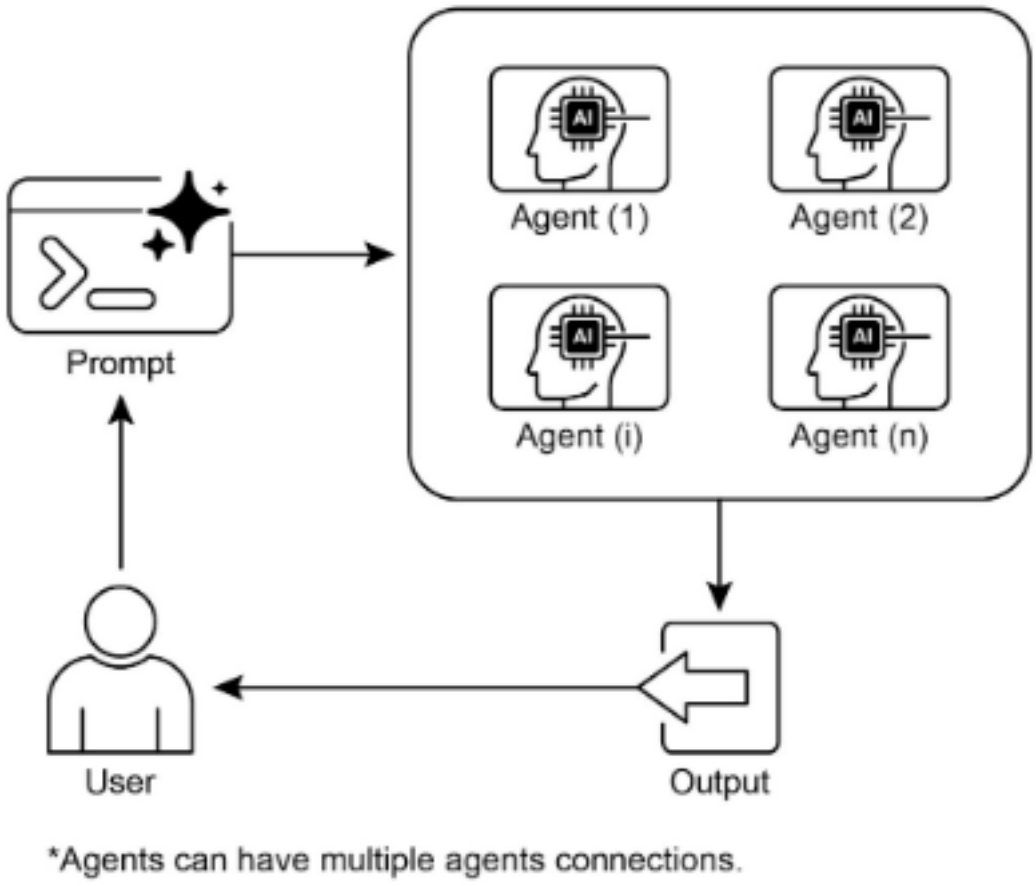

Figure 16: Multi–Agent Design Pattern

Fig.3: Multi–Agent design pattern

圖 3：多代理人設計模式

Key Takeaways

重點整理

- Multi–agent collaboration involves multiple agents working together to achieve a common goal.

- This pattern leverages specialized roles, distributed tasks, and inter-agent communication.

- Collaboration can take forms like sequential handoffs, parallel processing, debate, or hierarchical structures.

- This pattern is ideal for complex problems requiring diverse expertise or multiple distinct stages.
- 多代理人協作是多個代理人為共同目標協同工作。

- 此模式運用專門角色、分散任務與代理人間溝通。

- 協作形式包含序列交接、平行處理、辯論或階層結構。

- 此模式適合需要多元專業或多階段的複雜問題。

Conclusion

結論

This chapter explored the Multi-Agent Collaboration pattern, demonstrating the benefits of orchestrating multiple specialized agents within systems. We examined various collaboration models, emphasizing the pattern's essential role in addressing complex, multifaceted problems across diverse domains. Understanding agent collaboration naturally leads to an inquiry into their interactions with the external environment.

本章探討多代理人協作模式，展示在系統中協調多個專門代理人的效益。我們檢視了多種協作模型，強調此模式在跨領域處理複雜問題中的關鍵角色。理解代理人協作後，自然會進一步探問它們與外部環境的互動方式。

References

參考資料

1. Multi-Agent Collaboration Mechanisms: A Survey of LLMs, https://arxiv.org/abs/2501.06322
2. Multi-Agent System —The Power of Collaboration, https://aravindakumar.medium.com/introducing-multi-agent-frameworks-the-power-of-collaboration-e9db31bba1b6