02-Part_Two (Bilingual)

02-Part_Two（中英對照）

Chapter 8: Memory Management

第 8 章：記憶管理

Effective memory management is crucial for intelligent agents to retain information. Agents require different types of memory, much like humans, to operate efficiently. This chapter delves into memory management, specifically addressing the immediate (short-term) and persistent (long-term) memory requirements of agents.

有效的記憶管理對智慧代理人保留資訊至關重要。代理人需要不同類型的記憶，和人類相似，才能高效運作。本章深入探討記憶管理，特別說明代理人的即時（短期）與持久（長期）記憶需求。

In agent systems, memory refers to an agent's ability to retain and utilize information from past interactions, observations, and learning experiences. This capability allows agents to make informed decisions, maintain conversational context, and improve over time. Agent memory is generally categorized into two main types:

在代理系統中，記憶指代理人保留並運用過往互動、觀察與學習經驗資訊的能力。這使代理人能做出更明智的決策、維持對話脈絡並隨時間改進。代理人的記憶通常分為兩大類：

- **Short-Term Memory (Contextual Memory):** Similar to working memory, this holds information currently being processed or recently accessed. For agents using large language models (LLMs), short-term memory primarily exists within the context window. This window contains recent messages, agent replies, tool usage results, and agent reflections from the current interaction, all of which inform the LLM's subsequent responses and actions. The context window has a limited capacity, restricting the amount of recent information an agent can directly access. Efficient short-term memory management involves keeping the most relevant information within this limited space, possibly through techniques like summarizing older conversation segments or emphasizing key details. The advent of models with 'long context' windows

simply expands the size of this short-term memory, allowing more information to be held within a single interaction. However, this context is still ephemeral and is lost once the session concludes, and it can be costly and inefficient to process every time. Consequently, agents require separate memory types to achieve true persistence, recall information from past interactions, and build a lasting knowledge base.

- **Long-Term Memory (Persistent Memory):** This acts as a repository for information agents need to retain across various interactions, tasks, or extended periods, akin to long-term knowledge bases. Data is typically stored outside the agent's immediate processing environment, often in databases, knowledge graphs, or vector databases. In vector databases, information is converted into numerical vectors and stored, enabling agents to retrieve data based on semantic similarity rather than exact keyword matches, a process known as semantic search. When an agent needs information from long-term memory, it queries the external storage, retrieves relevant data, and integrates it into the short-term context for immediate use, thus combining prior knowledge with the current interaction.
- **短期記憶（情境記憶）：** 類似工作記憶，用於保存目前正在處理或剛存取的資訊。對使用大型語言模型（LLM）的代理人而言，短期記憶主要存在於上下文視窗中。此視窗包含近期訊息、代理回覆、工具使用結果與本次互動中的反思，這些都會影響 LLM 接下來的回應與行動。上下文視窗容量有限，限制代理能直接存取的近期資訊量。有效的短期記憶管理是在有限空間中保留最相關資訊，可能透過摘要較舊的對話片段或強調關鍵細節等方式。具備「長上下文」視窗的模型只是擴大短期記憶的大小，讓單次互動可容納更多資訊。然而這些脈絡仍是短暫的，會在會話結束後消失，且每次處理都可能昂貴且低效。因此，代理人需要額外的記憶類型以達成真正的持久性、回想過去互動資訊並建立長期知識庫。
- **長期記憶（持久記憶）：** 作為代理需要在多次互動、任務或長時間內保留資訊的儲存庫，類似長期知識庫。資料通常存放在代理的即時處理環境之外，例如資料庫、知識圖譜或向量資料庫。在向量資料庫中，資訊會被轉換為數值向量並儲存，讓代理能以語意相似度而非關鍵字精確匹配來檢索資料，這個過程稱為語意搜尋。當代理需要長期記憶中的資訊時，它會查詢外部儲存，取回相關資料並整合到短期脈絡中以便即時使用，進而將既有知識與當前互動結合。

Practical Applications & Use Cases

實務應用與使用情境

Memory management is vital for agents to track information and perform intelligently over time. This is essential for agents to surpass basic question-answering capabilities. Applications include:

記憶管理對代理人追蹤資訊並長期做出智慧表現至關重要，也是讓代理超越基本問答能力的關鍵。應用包含：

- **Chatbots and Conversational AI:** Maintaining conversation flow relies on short-term memory. Chatbots require remembering prior user inputs to provide coherent responses. Long-term memory enables chatbots to recall user preferences, past issues, or prior discussions, offering personalized and continuous interactions.
- **Task-Oriented Agents:** Agents managing multi-step tasks need short-term memory to track previous steps, current progress, and overall goals. This information might reside in the task's context or temporary storage. Long-term memory is crucial for accessing specific user-related data not in the immediate context.
- **Personalized Experiences:** Agents offering tailored interactions utilize long-term memory to store and retrieve user preferences, past behaviors, and personal information. This allows agents to adapt their responses and suggestions.
- **Learning and Improvement:** Agents can refine their performance by learning from past interactions. Successful strategies, mistakes, and new information are stored in long-term memory, facilitating future adaptations. Reinforcement learning agents store learned strategies or knowledge in this way.
- **Information Retrieval (RAG):** Agents designed for answering questions access a knowledge base, their long-term memory, often implemented within Retrieval Augmented Generation (RAG). The agent retrieves relevant documents or data to inform its responses.
- **Autonomous Systems:** Robots or self-driving cars require memory for maps, routes, object locations, and learned behaviors. This involves short-term

memory for immediate surroundings and long-term memory for general environmental knowledge.

- **聊天機器人與對話式 AI：** 維持對話流暢依賴短期記憶。聊天機器人需要記住先前的使用者輸入才能給出一致回應。長期記憶讓聊天機器人能回想使用者偏好、過往問題或之前的對話，提供個人化且連續的互動。
- **任務導向代理：** 管理多步驟任務的代理需要短期記憶來追蹤先前步驟、目前進度與整體目標。這些資訊可能存在於任務脈絡或暫存空間中。長期記憶則對存取不在即時脈絡中的特定使用者資料至關重要。
- **個人化體驗：** 提供客製互動的代理會使用長期記憶來儲存與取回使用者偏好、過往行為與個人資訊，讓代理能調整回應與建議。
- **學習與改進：** 代理能從過往互動中學習以改進表現。成功策略、錯誤與新資訊會存入長期記憶，促進未來調整。強化學習代理也會以此方式儲存策略或知識。
- **資訊檢索（RAG）：** 用於回答問題的代理會存取知識庫（即長期記憶），常透過檢索增強生成（RAG）實作。代理會取回相關文件或資料以支撐回應。
- **自主系統：** 機器人或自駕車需要記憶地圖、路徑、物體位置與已學習行為，這涉及用短期記憶掌握即時環境，並用長期記憶保存一般環境知識。

Memory enables agents to maintain history, learn, personalize interactions, and manage complex, time-dependent problems.

記憶讓代理人能保存歷史、學習、個人化互動，並處理複雜且具時間依賴性的問題。

Hands-On Code: Memory Management in Google Agent Developer Kit (ADK)

實作程式碼：Google Agent Developer Kit（ADK）的記憶管理

The Google Agent Developer Kit (ADK) offers a structured method for managing context and memory, including components for practical application. A solid grasp of ADK's Session, State, and Memory is vital for building agents that need to retain information.

Google Agent Developer Kit（ADK）提供一套結構化方法來管理上下文與記憶，並包含實作元件。要打造需要保留資訊的代理人，必須深入理解 ADK 的 Session、State 與 Memory。

Just as in human interactions, agents require the ability to recall previous exchanges to conduct coherent and natural conversations. ADK simplifies context management through three core concepts and their associated services.

如同人類互動，代理人需要能回想先前交流，才能進行連貫且自然的對話。ADK 透過三個核心概念與其服務，簡化了上下文管理。

Every interaction with an agent can be considered a unique conversation thread. Agents might need to access data from earlier interactions. ADK structures this as follows:

每次與代理人的互動都可視為一條獨立的對話脈絡。代理可能需要存取先前互動的資料。ADK 的結構如下：

- **Session:** An individual chat thread that logs messages and actions (Events) for that specific interaction, also storing temporary data (State) relevant to that conversation.
- **State (`session.state`):** Data stored within a Session, containing information relevant only to the current, active chat thread.
- **Memory:** A searchable repository of information sourced from various past chats or external sources, serving as a resource for data retrieval beyond the immediate conversation.
- **Session：** 一個獨立的聊天執行緒，會記錄該次互動的訊息與行動（Events），也會儲存與該對話相關的暫存資料（State）。
- **State（`session.state`）：** 存在 Session 內的資料，只與當前活躍的對話脈絡相關。
- **Memory：** 可搜尋的資訊儲存庫，來源於各種過往對話或外部資料，用於在即時對話之外進行資料檢索。

ADK provides dedicated services for managing critical components essential for building complex, stateful, and context-aware agents. The SessionService manages chat threads (Session objects) by handling their initiation, recording, and termination, while the MemoryService oversees the storage and retrieval of long-term knowledge (Memory).

ADK 提供專用服務來管理建構複雜、有狀態且具上下文感知代理所需的關鍵元件。SessionService 負責管理聊天執行緒（Session 物件）的建立、紀錄與終止，而 MemoryService 負責長期知識（Memory）的儲存與檢索。

Both the SessionService and MemoryService offer various configuration options, allowing users to choose storage methods based on application needs. In-memory options are available for testing purposes, though data will not persist across restarts. For persistent storage and scalability, ADK also supports database and cloud-based services.

SessionService 與 MemoryService 都提供多種設定選項，讓使用者可依應用需求選擇儲存方式。可用記憶體內存作測試，但資料不會在重啟後保留。若需持久化與可擴展性，ADK 也支援資料庫與雲端服務。

Session: Keeping Track of Each Chat

Session：追蹤每次對話

A Session object in ADK is designed to track and manage individual chat threads. Upon initiation of a conversation with an agent, the SessionService generates a Session object, represented as `google.adk.sessions.Session`. This object encapsulates all data relevant to a specific conversation thread, including unique identifiers (`id`, `app_name`, `user_id`), a chronological record of events as Event objects, a storage area for session-specific temporary data known as state, and a timestamp indicating the last update (`last_update_time`). Developers typically interact with Session objects indirectly through the SessionService. The SessionService is responsible for managing the lifecycle of conversation sessions, which includes initiating new sessions, resuming previous sessions, recording session activity (including state updates), identifying active sessions, and managing the removal of session data. The ADK provides several SessionService implementations with varying storage mechanisms for session history and temporary data, such as the InMemorySessionService, which is suitable for testing but does not provide data persistence across application restarts.

ADK 中的 Session 物件用於追蹤與管理單一聊天執行緒。當與代理人開始對話時，SessionService 會產生一個 Session 物件（`google.adk.sessions.Session`）。該物件包含特定對話脈絡的所有資料，包括唯一識別碼（`id`、`app_name`、`user_id`）、事件的時間序列紀錄（Event 物件）、稱為 state 的對話暫存資料區，以及最後更新時間戳（`last_update_time`）。開發者通常透過 SessionService 間接與 Session 互動。SessionService 負責管理會話生命週期，包括啟動新會話、恢復舊會話、記錄會話活動（含 state 更新）、辨識活躍會話，以及移除會話資料。ADK 提供多種

SessionService 實作，使用不同的儲存機制保存會話歷史與暫存資料，例如 InMemorySessionService 適合測試但不會在重啟後保留資料。

```
# Example: Using InMemorySessionService
# This is suitable for local development and testing where data
# persistence across application restarts is not required.
from google.adk.sessions import InMemorySessionService
session_service = InMemorySessionService()
```

Then there's DatabaseSessionService if you want reliable saving to a database you manage.

如果你需要可靠地儲存到自己管理的資料庫，則可使用 DatabaseSessionService。

```
# Example: Using DatabaseSessionService
# This is suitable for production or development requiring persistent
storage.
# You need to configure a database URL (e.g., for SQLite, PostgreSQL, etc.).
# Requires: pip install google-adk[sqlalchemy] and a database driver (e.g.,
psycopg2 for PostgreSQL)
from google.adk.sessions import DatabaseSessionService
# Example using a local SQLite file:
db_url = "sqlite:///./my_agent_data.db"
session_service = DatabaseSessionService(db_url=db_url)
```

Besides, there's VertexAiSessionService which uses Vertex AI infrastructure for scalable production on Google Cloud.

此外，還有使用 Vertex AI 基礎架構的 VertexAiSessionService，適用於 Google Cloud 上可擴展的正式環境。

```
# Example: Using VertexAiSessionService
# This is suitable for scalable production on Google Cloud Platform,
leveraging
# Vertex AI infrastructure for session management.
# Requires: pip install google-adk[vertexai] and GCP setup/authentication

from google.adk.sessions import VertexAiSessionService


PROJECT_ID = "your-gcp-project-id"  # Replace with your GCP project ID
LOCATION = "us-central1"  # Replace with your desired GCP location

# The app_name used with this service should correspond to the Reasoning
Engine ID or name
REASONING_ENGINE_APP_NAME = (
    "projects/your-gcp-project-id/locations/us-central1/reasoningEngines/your-
engine-id"
```

```
)  # Replace with your Reasoning Engine resource name

session_service = VertexAiSessionService(project=PROJECT_ID,
location=LOCATION)

# When using this service, pass REASONING_ENGINE_APP_NAME to service methods:
# session_service.create_session(app_name=REASONING_ENGINE_APP_NAME, ...)
# session_service.get_session(app_name=REASONING_ENGINE_APP_NAME, ...)
# session_service.append_event(session, event,
app_name=REASONING_ENGINE_APP_NAME)
# session_service.delete_session(app_name=REASONING_ENGINE_APP_NAME, ...)
```

Choosing an appropriate SessionService is crucial as it determines how the agent's interaction history and temporary data are stored and their persistence.

選擇合適的 SessionService 至關重要，因為它決定了代理人的互動歷史與暫存資料如何儲存，以及是否能持久保存。

Each message exchange involves a cyclical process: A message is received, the Runner retrieves or establishes a Session using the SessionService, the agent processes the message using the Session's context (state and historical interactions), the agent generates a response and may update the state, the Runner encapsulates this as an Event, and the `session_service.append_event` method records the new event and updates the state in storage. The Session then awaits the next message. Ideally, the `delete_session` method is employed to terminate the session when the interaction concludes. This process illustrates how the SessionService maintains continuity by managing the Session-specific history and temporary data.

每次訊息交換都會經歷一個循環流程：接收訊息、Runner 透過 SessionService 取得或建立 Session、代理用 Session 的脈絡（state 與歷史互動）處理訊息、代理產生回應並可能更新 state、Runner 將其封裝為 Event，然後 `session_service.append_event` 會記錄新事件並更新儲存中的 state。Session 接著等待下一則訊息。理想情況下，互動結束時會呼叫 `delete_session` 來終止會話。此流程說明 SessionService 如何透過管理 Session 的歷史與暫存資料維持連續性。

State: The Session's Scratchpad

State：Session 的草稿紙

In the ADK, each Session, representing a chat thread, includes a state component akin to an agent's temporary working memory for the duration of that specific conversation. While session.events logs the entire chat history,

session.state stores and updates dynamic data points relevant to the active chat.

在 ADK 中，每個 Session（代表一條對話執行緒）都包含 state 元件，類似代理在該次對話期間的暫時工作記憶。雖然 session.events 記錄整段對話歷史，session.state 則儲存並更新與當前對話相關的動態資料點。

Fundamentally, session.state operates as a dictionary, storing data as key–value pairs. Its core function is to enable the agent to retain and manage details essential for coherent dialogue, such as user preferences, task progress, incremental data collection, or conditional flags influencing subsequent agent actions.

從本質上來說，session.state 以字典方式運作，使用鍵值對儲存資料。其核心功能是讓代理保留並管理對話所需的重要細節，例如使用者偏好、任務進度、逐步蒐集的資料或影響後續行動的條件旗標。

The state's structure comprises string keys paired with values of serializable Python types, including strings, numbers, booleans, lists, and dictionaries containing these basic types. State is dynamic, evolving throughout the conversation. The permanence of these changes depends on the configured SessionService.

state 的結構由字串鍵與可序列化的 Python 型別值組成，包括字串、數字、布林、列表，以及包含這些基本型別的字典。state 是動態的，會隨對話演進而改變。這些變更是否持久保存取決於設定的 SessionService。

State organization can be achieved using key prefixes to define data scope and persistence. Keys without prefixes are session–specific.

可透過鍵前綴來組織 state，以定義資料範圍與持久性。沒有前綴的鍵為 Session 專用。

- The user: prefix associates data with a user ID across all sessions.
- The app: prefix designates data shared among all users of the application.
- The temp: prefix indicates data valid only for the current processing turn and is not persistently stored.
- user: 前綴將資料與某使用者 ID 跨所有 Session 關聯。
- app: 前綴表示資料在應用程式所有使用者之間共享。
- temp: 前綴表示資料只在當前處理回合有效，且不會持久保存。

The agent accesses all state data through a single session.state dictionary. The SessionService handles data retrieval, merging, and persistence. State should be updated upon adding an Event to the session history via `session_service.append_event()`. This ensures accurate tracking, proper saving in persistent services, and safe handling of state changes.

代理透過單一的 session.state 字典存取所有 state 資料。SessionService 負責資料取得、合併與持久化。應在透過 `session_service.append_event()` 將 Event 加入會話歷史時更新 state，以確保正確追蹤、在持久化服務中妥善儲存，以及安全處理 state 變更。

1. The Simple Way: Using `output_key` (for Agent Text Replies)

1. 簡單方式：使用 `output_key`（用於代理文字回覆）

This is the easiest method if you just want to save your agent's final text response directly into the state. When you set up your LlmAgent, just tell it the output_key you want to use. The Runner sees this and automatically creates the necessary actions to save the response to the state when it appends the event. Let's look at a code example demonstrating state update via `output_key`.

如果你只想把代理的最終文字回覆直接存入 state，這是最簡單的方法。設定 LlmAgent 時只要指定要用的 output_key。Runner 看到後會在附加事件時自動建立必要動作，把回覆存入 state。以下示範透過 `output_key` 更新 state 的範例。

```
# Import necessary classes from the Google Agent Developer Kit (ADK)
from google.adk.agents import LlmAgent
from google.adk.sessions import InMemorySessionService, Session
from google.adk.runners import Runner
from google.genai.types import Content, Part


# Define an LlmAgent with an output_key.
greeting_agent = LlmAgent(
 name="Greeter",
 model="gemini-2.0-flash",
 instruction="Generate a short, friendly greeting.",
 output_key="last_greeting",
)


# --- Setup Runner and Session ---
app_name, user_id, session_id = "state_app", "user1", "session1"
```

```
session_service = InMemorySessionService()

runner = Runner(
    agent=greeting_agent,
    app_name=app_name,
    session_service=session_service,
)

session = session_service.create_session(
    app_name=app_name,
    user_id=user_id,
    session_id=session_id,
)

print(f"Initial state: {session.state}")


# --- Run the Agent ---
user_message = Content(parts=[Part(text="Hello")])

print("\n--- Running the agent ---")
for event in runner.run(
    user_id=user_id,
    session_id=session_id,
    new_message=user_message,
):
    if event.is_final_response():
        print("Agent responded.")


# --- Check Updated State ---
# Correctly check the state after the runner has finished processing all
events.
updated_session = session_service.get_session(app_name, user_id, session_id)
print(f"\nState after agent run: {updated_session.state}")
```

Behind the scenes, the Runner sees your `output_key` and automatically creates the necessary actions with a `state_delta` when it calls `append_event`.

在幕後，Runner 看到你的 `output_key` 後，會在呼叫 `append_event` 時自動建立帶有 `state_delta` 的必要動作。

2. The Standard Way: Using `EventActions.state_delta` (for More Complicated Updates)

2. 標準方式：使用 `EventActions.state_delta`（用於更複雜的更新）

For times when you need to do more complex things — like updating several keys at once, saving things that aren't just text, targeting specific scopes like user: or app:, or making updates that aren't tied to the agent's final text reply — you'll manually build a dictionary of your state changes (the `state_delta`) and include it within the EventActions of the Event you're appending. Let's look at one example:

當你需要做更複雜的事情時，例如一次更新多個鍵、保存非文字內容、鎖定 user: 或 app: 等特定範圍，或進行與代理最終文字回覆無關的更新，你就需要手動建立 state 變更字典（`state_delta`），並將其加入要附加的 Event 的 EventActions 中。以下是一個範例：

```
import time

from google.adk.tools.tool_context import ToolContext
from google.adk.sessions import InMemorySessionService


# --- Define the Recommended Tool-Based Approach ---
def log_user_login(tool_context: ToolContext) -> dict:
    """
    Updates the session state upon a user login event.
    This tool encapsulates all state changes related to a user login.

    Args:
        tool_context: Automatically provided by ADK, gives access to session
state.

    Returns:
        A dictionary confirming the action was successful.
    """
    # Access the state directly through the provided context.
    state = tool_context.state

    # Get current values or defaults, then update the state.
    # This is much cleaner and co-locates the logic.
    login_count = state.get("user:login_count", 0) + 1
    state["user:login_count"] = login_count
    state["task_status"] = "active"
    state["user:last_login_ts"] = time.time()
```

```
    state["temp:validation_needed"] = True

    print("State updated from within the `log_user_login` tool.")

    return {
        "status": "success",
        "message": f"User login tracked. Total logins: {login_count}.",
    }


# --- Demonstration of Usage ---
# In a real application, an LLM Agent would decide to call this tool.
# Here, we simulate a direct call for demonstration purposes.

# 1. Setup
session_service = InMemorySessionService()
app_name, user_id, session_id = "state_app_tool", "user3", "session3"

session = session_service.create_session(
    app_name=app_name,
    user_id=user_id,
    session_id=session_id,
    state={"user:login_count": 0, "task_status": "idle"},
)

print(f"Initial state: {session.state}")

# 2. Simulate a tool call (in a real app, the ADK Runner does this)
# We create a ToolContext manually just for this standalone example.
from google.adk.tools.tool_context import InvocationContext

mock_context = ToolContext(
    invocation_context=InvocationContext(
        app_name=app_name,
        user_id=user_id,
        session_id=session_id,
        session=session,
        session_service=session_service,
    )
)

# 3. Execute the tool
log_user_login(mock_context)

# 4. Check the updated state
updated_session = session_service.get_session(app_name, user_id, session_id)
print(f"State after tool execution: {updated_session.state}")
```

```
# Expected output will show the same state change as the "Before" case,
# but the code organization is significantly cleaner and more robust.
```

This code demonstrates a tool-based approach for managing user session state in an application. It defines a function log_user_login, which acts as a tool. This tool is responsible for updating the session state when a user logs in.
The function takes a ToolContext object, provided by the ADK, to access and modify the session's state dictionary. Inside the tool, it increments a user:login_count, sets the task_status to "active", records the user:last_login_ts (timestamp), and adds a temporary flag temp:validation_needed.

這段程式碼示範在應用程式中以工具為基礎管理使用者 Session 狀態的方法。它定義了 log_user_login 函式作為工具，負責在使用者登入時更新 Session 狀態。
該函式透過 ADK 提供的 ToolContext 物件存取並修改 Session 的 state 字典。在工具內部，它會遞增 user:login_count、把 task_status 設為 "active"、記錄 user:last_login_ts（時間戳），並加入臨時旗標 temp:validation_needed。

The demonstration part of the code simulates how this tool would be used. It sets up an in-memory session service and creates an initial session with some predefined state. A ToolContext is then manually created to mimic the environment in which the ADK Runner would execute the tool. The `log_user_login` function is called with this mock context. Finally, the code retrieves the session again to show that the state has been updated by the tool's execution. The goal is to show how encapsulating state changes within tools makes the code cleaner and more organized compared to directly manipulating state outside of tools.

程式碼的示範部分模擬了此工具的使用方式。它建立記憶體內 Session 服務並以預設 state 建立初始 Session。接著手動建立 ToolContext 以模擬 ADK Runner 執行工具的環境，再以該 mock context 呼叫 `log_user_login`。最後再次取得 Session 以顯示 state 已被更新。重點在於示範將 state 變更封裝在工具內，會比直接在工具外操作 state 更乾淨、組織更清楚。

Note that direct modification of the `session.state` dictionary after retrieving a session is strongly discouraged as it bypasses the standard event processing mechanism. Such direct changes will not be recorded in the session's event history, may not be persisted by the selected `SessionService`, could lead to concurrency issues, and will not update essential metadata such as timestamps.

The recommended methods for updating the session state are using the `output_key` parameter on an `LlmAgent` (specifically for the agent's final text responses) or including state changes within `EventActions.state_delta` when appending an event via `session_service.append_event()`. The `session.state` should primarily be used for reading existing data.

請注意，在取得 Session 後直接修改 `session.state` 字典是強烈不建議的，因為這會繞過標準事件處理機制。這種直接變更不會記錄在 Session 的事件歷史中，可能無法被所選的 `SessionService` 持久化，可能引發並行問題，也不會更新像是時間戳等重要中繼資料。建議的更新方式是：在 `LlmAgent` 上使用 `output_key`（用於代理的最終文字回覆），或在透過 `session_service.append_event()` 附加事件時，將 state 變更放在 `EventActions.state_delta` 中。`session.state` 主要用於讀取現有資料。

To recap, when designing your state, keep it simple, use basic data types, give your keys clear names and use prefixes correctly, avoid deep nesting, and always update state using the append_event process.

總結來說，設計 state 時要保持簡潔，使用基本資料型別，為鍵取清楚名稱並正確使用前綴，避免深層巢狀，並一律透過 append_event 流程更新 state。

Memory: Long-Term Knowledge with MemoryService

Memory：透過 MemoryService 的長期知識

In agent systems, the Session component maintains a record of the current chat history (events) and temporary data (state) specific to a single conversation. However, for agents to retain information across multiple interactions or access external data, long-term knowledge management is necessary. This is facilitated by the MemoryService.

在代理系統中，Session 元件維護當前對話歷史（events）與該次對話專用的暫存資料（state）。然而，若代理要在多次互動間保留資訊或存取外部資料，就需要長期知識管理，而這由 MemoryService 提供支援。

```
# Example: Using InMemoryMemoryService
# This is suitable for local development and testing where data
# persistence across application restarts is not required.
# Memory content is lost when the app stops.

from google.adk.memory import InMemoryMemoryService

memory_service = InMemoryMemoryService()
```

Session and State can be conceptualized as short-term memory for a single chat session, whereas the Long-Term Knowledge managed by the MemoryService functions as a persistent and searchable repository. This repository may contain information from multiple past interactions or external sources. The MemoryService, as defined by the BaseMemoryService interface, establishes a standard for managing this searchable, long-term knowledge. Its primary functions include adding information, which involves extracting content from a session and storing it using the add_session_to_memory method, and retrieving information, which allows an agent to query the store and receive relevant data using the search_memory method.

Session 與 State 可視為單一對話的短期記憶，而 MemoryService 管理的長期知識則是可持久且可搜尋的儲存庫。此儲存庫可包含多次過往互動或外部來源的資訊。由 BaseMemoryService 介面定義的 MemoryService，建立了管理此可搜尋長期知識的標準。其主要功能包括新增資訊（從 Session 中擷取內容並透過 add_session_to_memory 儲存）與檢索資訊（代理透過 search_memory 查詢儲存庫並取得相關資料）。

The ADK offers several implementations for creating this long-term knowledge store. The InMemoryMemoryService provides a temporary storage solution suitable for testing purposes, but data is not preserved across application restarts. For production environments, the VertexAiRagMemoryService is typically utilized. This service leverages Google Cloud's Retrieval Augmented Generation (RAG) service, enabling scalable, persistent, and semantic search capabilities (Also, refer to the chapter 14 on RAG).

ADK 提供多種實作來建立長期知識儲存庫。InMemoryMemoryService 提供適合測試的暫時性儲存方案，但資料不會在應用重啟後保留。正式環境通常使用 VertexAiRagMemoryService。此服務運用 Google Cloud 的檢索增強生成（RAG）服務，提供可擴展、持久且具語意搜尋能力的儲存（另可參考第 14 章 RAG）。

```
# Example: Using VertexAiRagMemoryService
# This is suitable for scalable production on GCP, leveraging
# Vertex AI RAG (Retrieval Augmented Generation) for persistent,
# searchable memory.
# Requires: pip install google-adk[vertexai], GCP
# setup/authentication, and a Vertex AI RAG Corpus.

from google.adk.memory import VertexAiRagMemoryService
```

```
# The resource name of your Vertex AI RAG Corpus
RAG_CORPUS_RESOURCE_NAME = (
    "projects/your-gcp-project-id/locations/us-central1/ragCorpora/your-
corpus-id"
)  # Replace with your Corpus resource name

# Optional configuration for retrieval behavior
SIMILARITY_TOP_K = 5  # Number of top results to retrieve
VECTOR_DISTANCE_THRESHOLD = 0.7  # Threshold for vector similarity

memory_service = VertexAiRagMemoryService(
    rag_corpus=RAG_CORPUS_RESOURCE_NAME,
    similarity_top_k=SIMILARITY_TOP_K,
    vector_distance_threshold=VECTOR_DISTANCE_THRESHOLD,
)

# When using this service, methods like add_session_to_memory
# and search_memory will interact with the specified Vertex AI
# RAG Corpus.
```

Hands-on code: Memory Management in LangChain and LangGraph

實作程式碼：LangChain 與 LangGraph 的記憶管理

In LangChain and LangGraph, Memory is a critical component for creating intelligent and natural-feeling conversational applications. It allows an AI agent to remember information from past interactions, learn from feedback, and adapt to user preferences. LangChain's memory feature provides the foundation for this by referencing a stored history to enrich current prompts and then recording the latest exchange for future use. As agents handle more complex tasks, this capability becomes essential for both efficiency and user satisfaction.

在 LangChain 與 LangGraph 中，Memory 是建立智慧且自然對話應用的關鍵元件。它讓 AI 代理能記住過去互動資訊、從回饋中學習，並適應使用者偏好。LangChain 的記憶功能透過引用已儲存的歷史來強化當前提示，並將最新對話記錄供日後使用。當代理面對更複雜的任務時，此能力對效率與使用者滿意度都至關重要。

Short-Term Memory: This is thread-scoped, meaning it tracks the ongoing conversation within a single session or thread. It provides immediate context, but a full history can challenge an LLM's context window, potentially leading to

errors or poor performance. LangGraph manages short-term memory as part of the agent's state, which is persisted via a checkpointer, allowing a thread to be resumed at any time.

短期記憶： 這是執行緒範圍的記憶，追蹤單一 Session 或執行緒內的進行中對話。它提供即時脈絡，但完整歷史可能會擠壓 LLM 的上下文視窗，導致錯誤或效能下降。LangGraph 將短期記憶視為代理 state 的一部分，並透過 checkpointer 持久化，使執行緒可在任何時間被恢復。

Long-Term Memory: This stores user-specific or application-level data across sessions and is shared between conversational threads. It is saved in custom "namespaces" and can be recalled at any time in any thread. LangGraph provides stores to save and recall long-term memories, enabling agents to retain knowledge indefinitely.

長期記憶： 它在多個 Session 之間保存使用者特定或應用層級資料，並可在對話執行緒間共享。資料被儲存在自訂「命名空間」中，並可在任何執行緒中隨時取回。LangGraph 提供 store 來儲存與取回長期記憶，讓代理能無限期保留知識。

LangChain provides several tools for managing conversation history, ranging from manual control to automated integration within chains.

LangChain 提供多種工具管理對話歷史，從手動控制到在 chain 中的自動整合都有。

ChatMessageHistory: Manual Memory Management. For direct and simple control over a conversation's history outside of a formal chain, the ChatMessageHistory class is ideal. It allows for the manual tracking of dialogue exchanges.

ChatMessageHistory：手動記憶管理。 若需在正式 chain 之外對對話歷史進行直接且簡單的控制，ChatMessageHistory 類別最適合。它可手動追蹤對話交換。

```
from langchain.memory import ChatMessageHistory


# Initialize the history object
history = ChatMessageHistory()

# Add user and AI messages
history.add_user_message("I'm heading to New York next week.")
history.add_ai_message("Great! It's a fantastic city.")

# Access the list of messages
print(history.messages)
```

ConversationBufferMemory: Automated Memory for Chains. For integrating memory directly into chains, ConversationBufferMemory is a common choice. It holds a buffer of the conversation and makes it available to your prompt. Its behavior can be customized with two key parameters:

ConversationBufferMemory：用於 chain 的自動記憶。 若要將記憶直接整合進 chain，ConversationBufferMemory 是常見選擇。它保留對話緩衝並提供給提示使用。其行為可透過兩個重要參數自訂：

- `memory_key`: A string that specifies the variable name in your prompt that will hold the chat history. It defaults to “history”.
- `return_messages`: A boolean that dictates the format of the history.
  - If `False` (the default), it returns a single formatted string, which is ideal for standard LLMs.
  - If `True`, it returns a list of message objects, which is the recommended format for Chat Models.
- `memory_key`：指定提示中用於保存聊天歷史的變數名稱字串，預設為“history”。
- `return_messages`：決定歷史格式的布林值。
  - 若為 `False`（預設），回傳單一格式化字串，適合標準 LLM。
  - 若為 `True`，回傳訊息物件列表，為聊天模型的建議格式。

```
from langchain.memory import ConversationBufferMemory


# Initialize memory
memory = ConversationBufferMemory()

# Save a conversation turn
memory.save_context(
    {"input": "What's the weather like?"},
    {"output": "It's sunny today."},
)

# Load the memory as a string
print(memory.load_memory_variables({}))
```

Integrating this memory into an LLMChain allows the model to access the conversation’s history and provide contextually relevant responses

將此記憶整合到 LLMChain 可讓模型存取對話歷史，並提供具脈絡的回應。

```
from langchain_openai import OpenAI
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate
```

```
from langchain.memory import ConversationBufferMemory


# 1. Define LLM and Prompt
llm = OpenAI(temperature=0)

template = """You are a helpful travel agent.
Previous conversation: {history}
New question: {question}
Response:"""
prompt = PromptTemplate.from_template(template)

# 2. Configure Memory
# The memory_key "history" matches the variable in the prompt
memory = ConversationBufferMemory(memory_key="history")

# 3. Build the Chain
conversation = LLMChain(llm=llm, prompt=prompt, memory=memory)

# 4. Run the Conversation
response = conversation.predict(question="I want to book a flight.")
print(response)

response = conversation.predict(question="My name is Sam, by the way.")
print(response)

response = conversation.predict(question="What was my name again?")
print(response)
```

For improved effectiveness with chat models, it is recommended to use a structured list of message objects by setting `return_messages=True`.

為了提升聊天模型的效果，建議設定 `return_messages=True` 以使用結構化的訊息物件列表。

```
from langchain_openai import ChatOpenAI
from langchain.chains import LLMChain
from langchain.memory import ConversationBufferMemory
from langchain_core.prompts import (
    ChatPromptTemplate,
    MessagesPlaceholder,
    SystemMessagePromptTemplate,
    HumanMessagePromptTemplate,
)


# 1. Define Chat Model and Prompt
```

```
llm = ChatOpenAI()

prompt = ChatPromptTemplate(
    messages=[
        SystemMessagePromptTemplate.from_template("You are a friendly
assistant."),
        MessagesPlaceholder(variable_name="chat_history"),
        HumanMessagePromptTemplate.from_template("{question}"),
    ]
)

# 2. Configure Memory
# return_messages=True is essential for chat models
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True)

# 3. Build the Chain
conversation = LLMChain(llm=llm, prompt=prompt, memory=memory)

# 4. Run the Conversation
response = conversation.predict(question="Hi, I'm Jane.")
print(response)

response = conversation.predict(question="Do you remember my name?")
print(response)
```

Types of Long-Term Memory: Long-term memory allows systems to retain information across different conversations, providing a deeper level of context and personalization. It can be broken down into three types analogous to human memory:

長期記憶的類型： 長期記憶讓系統能跨不同對話保留資訊，提供更深層的脈絡與個人化。它可類比人類記憶分為三類：

- **Semantic Memory: Remembering Facts:** This involves retaining specific facts and concepts, such as user preferences or domain knowledge. It is used to ground an agent's responses, leading to more personalized and relevant interactions. This information can be managed as a continuously updated user "profile" (a JSON document) or as a "collection" of individual factual documents.
- **Episodic Memory: Remembering Experiences:** This involves recalling past events or actions. For AI agents, episodic memory is often used to remember how to accomplish a task. In practice, it's frequently implemented through

few-shot example prompting, where an agent learns from past successful interaction sequences to perform tasks correctly.

- **Procedural Memory: Remembering Rules:** This is the memory of how to perform tasks—the agent's core instructions and behaviors, often contained in its system prompt. It's common for agents to modify their own prompts to adapt and improve. An effective technique is "Reflection," where an agent is prompted with its current instructions and recent interactions, then asked to refine its own instructions.
- **語意記憶：記住事實。** 指保留特定事實與概念，例如使用者偏好或領域知識。它用來支撐代理的回應，使互動更個人化且相關。這些資訊可管理為持續更新的使用者「檔案」（JSON 文件）或個別事實文件的「集合」。
- **情節記憶：記住經驗。** 指回想過去事件或行動。對 AI 代理而言，情節記憶常用來記住如何完成任務。實作上常透過少量示例提示（few-shot）來達成，代理從過往成功互動序列中學習，以正確執行任務。
- **程序記憶：記住規則。** 這是如何執行任務的記憶，即代理的核心指令與行為，常存在於系統提示中。代理常會修改自己的提示來適應與改進。一個有效技巧是「反思（Reflection）」：讓代理面對自身指令與近期互動，再要求它修訂自己的指令。

Below is pseudo-code demonstrating how an agent might use reflection to update its procedural memory stored in a LangGraph BaseStore

以下是示意性程式碼，展示代理如何透過反思更新儲存在 LangGraph BaseStore 中的程序記憶。

```
# Node that updates the agent's instructions
def update_instructions(state: State, store: BaseStore):
    namespace = ("instructions",)

    # Get the current instructions from the store
    current_instructions = store.search(namespace)[0]

    # Create a prompt to ask the LLM to reflect on the conversation
    # and generate new, improved instructions
    prompt = prompt_template.format(
        instructions=current_instructions.value["instructions"],
        conversation=state["messages"],
    )

    # Get the new instructions from the LLM
    output = llm.invoke(prompt)
```

```
    new_instructions = output["new_instructions"]

    # Save the updated instructions back to the store
    store.put(("agent_instructions",), "agent_a", {"instructions":
new_instructions})


# Node that uses the instructions to generate a response
def call_model(state: State, store: BaseStore):
    namespace = ("agent_instructions",)

    # Retrieve the latest instructions from the store
    instructions = store.get(namespace, key="agent_a")[0]

    # Use the retrieved instructions to format the prompt
    prompt = prompt_template.format(
        instructions=instructions.value["instructions"]
    )
    # ... application logic continues
```

LangGraph stores long-term memories as JSON documents in a store. Each memory is organized under a custom namespace (like a folder) and a distinct key (like a filename). This hierarchical structure allows for easy organization and retrieval of information. The following code demonstrates how to use InMemoryStore to put, get, and search for memories.

LangGraph 將長期記憶以 JSON 文件形式儲存在 store 中。每段記憶都被組織在自訂命名空間（如資料夾）與獨立鍵（如檔名）之下。此階層式結構讓資訊易於組織與檢索。以下程式碼示範如何使用 InMemoryStore 來新增、取得與搜尋記憶。

```
from langgraph.store.memory import InMemoryStore


# A placeholder for a real embedding function
def embed(texts: list[str]) -> list[list[float]]:
    # In a real application, use a proper embedding model
    return [[1.0, 2.0] for _ in texts]


# Initialize an in-memory store. For production, use a database-backed store.
store = InMemoryStore(index={"embed": embed, "dims": 2})

# Define a namespace for a specific user and application context
user_id = "my-user"
application_context = "chitchat"
namespace = (user_id, application_context)
```

```
# 1. Put a memory into the store
store.put(
    namespace,
    "a-memory",  # The key for this memory
    {
        "rules": [
            "User likes short, direct language",
            "User only speaks English & python",
        ],
        "my-key": "my-value",
    },
)

# 2. Get the memory by its namespace and key
item = store.get(namespace, "a-memory")
print("Retrieved Item:", item)

# 3. Search for memories within the namespace, filtering by content
# and sorting by vector similarity to the query.
items = store.search(
    namespace,
    filter={"my-key": "my-value"},
    query="language preferences",
)
print("Search Results:", items)
```

Vertex Memory Bank

Vertex 記憶庫

Memory Bank, a managed service in the Vertex AI Agent Engine, provides agents with persistent, long-term memory. The service uses Gemini models to asynchronously analyze conversation histories to extract key facts and user preferences.

Memory Bank 是 Vertex AI Agent Engine 的受管理服務，為代理提供持久的長期記憶。該服務使用 Gemini 模型非同步分析對話歷史，萃取關鍵事實與使用者偏好。

This information is stored persistently, organized by a defined scope like user ID, and intelligently updated to consolidate new data and resolve contradictions. Upon starting a new session, the agent retrieves relevant memories through either a full data recall or a similarity search using embeddings. This process allows an agent to maintain continuity across sessions and personalize responses based on recalled information.

這些資訊會持久保存，並依使用者 ID 等定義範圍組織，且會智慧更新以整合新資料並解決矛盾。當啟動新 Session 時，代理會透過完整回想或以嵌入向量的相似度搜尋取回相關記憶。這個流程讓代理能跨 Session 維持連續性，並依回想資訊進行個人化回應。

The agent's runner interacts with the VertexAiMemoryBankService, which is initialized first. This service handles the automatic storage of memories generated during the agent's conversations. Each memory is tagged with a unique USER_ID and APP_NAME, ensuring accurate retrieval in the future.

代理的 runner 會與先初始化的 VertexAiMemoryBankService 互動。此服務負責自動儲存代理對話中產生的記憶。每段記憶都會標記唯一的 USER_ID 與 APP_NAME，以確保未來可準確取回。

```
from google.adk.memory import VertexAiMemoryBankService


agent_engine_id = agent_engine.api_resource.name.split("/")[-1]

memory_service = VertexAiMemoryBankService(
    project="PROJECT_ID",
    location="LOCATION",
    agent_engine_id=agent_engine_id,
)

session = await session_service.get_session(
    app_name=app_name,
    user_id="USER_ID",
    session_id=session.id,
)

await memory_service.add_session_to_memory(session)
```

Memory Bank offers seamless integration with the Google ADK, providing an immediate out-of-the-box experience. For users of other agent frameworks, such as LangGraph and CrewAI, Memory Bank also offers support through direct API calls. Online code examples demonstrating these integrations are readily available for interested readers.

Memory Bank 與 Google ADK 無縫整合，提供即刻可用的體驗。對於其他代理框架（如 LangGraph 與 CrewAI）的使用者，Memory Bank 也可透過直接 API 呼叫支援。線上也有示範這些整合的程式碼範例供讀者參考。

At a Glance

一覽

What: Agentic systems need to remember information from past interactions to perform complex tasks and provide coherent experiences. Without a memory mechanism, agents are stateless, unable to maintain conversational context, learn from experience, or personalize responses for users. This fundamentally limits them to simple, one-shot interactions, failing to handle multi-step processes or evolving user needs. The core problem is how to effectively manage both the immediate, temporary information of a single conversation and the vast, persistent knowledge gathered over time.

什麼： 代理式系統需要記住過去互動資訊，才能完成複雜任務並提供連貫體驗。沒有記憶機制，代理就會是無狀態的，無法維持對話脈絡、從經驗中學習，或為使用者提供個人化回應。這會使它們受限於一次性互動，無法處理多步驟流程或不斷變化的需求。核心問題在於如何有效管理單次對話的即時暫存資訊與隨時間累積的大量持久知識。

Why: The standardized solution is to implement a dual-component memory system that distinguishes between short-term and long-term storage. Short-term, contextual memory holds recent interaction data within the LLM's context window to maintain conversational flow. For information that must persist, long-term memory solutions use external databases, often vector stores, for efficient, semantic retrieval. Agentic frameworks like the Google ADK provide specific components to manage this, such as Session for the conversation thread and State for its temporary data. A dedicated MemoryService is used to interface with the long-term knowledge base, allowing the agent to retrieve and incorporate relevant past information into its current context.

為什麼： 標準做法是實作雙元記憶系統，區分短期與長期儲存。短期情境記憶在 LLM 的上下文視窗中保留近期互動資料，以維持對話流。對需持久保留的資訊，長期記憶通常使用外部資料庫（常為向量儲存）進行高效率的語意檢索。像 Google ADK 這類代理框架提供特定元件來管理這些，例如用 Session 表示對話執行緒、用 State 保存暫存資料；並以專用 MemoryService 介接長期知識庫，讓代理取回並整合相關過往資訊到當前脈絡中。

Rule of thumb: Use this pattern when an agent needs to do more than answer a single question. It is essential for agents that must maintain context throughout

a conversation, track progress in multi-step tasks, or personalize interactions by recalling user preferences and history. Implement memory management whenever the agent is expected to learn or adapt based on past successes, failures, or newly acquired information.

經驗法則： 當代理不只是回答單一問題時，就應使用此模式。若代理需在整段對話中維持脈絡、追蹤多步驟任務進度，或透過回想使用者偏好與歷史進行個人化互動，記憶管理就是必要的。當代理預期要從過去的成功、失敗或新取得資訊中學習或適應時，就應實作記憶管理。

Visual summary:

視覺摘要：

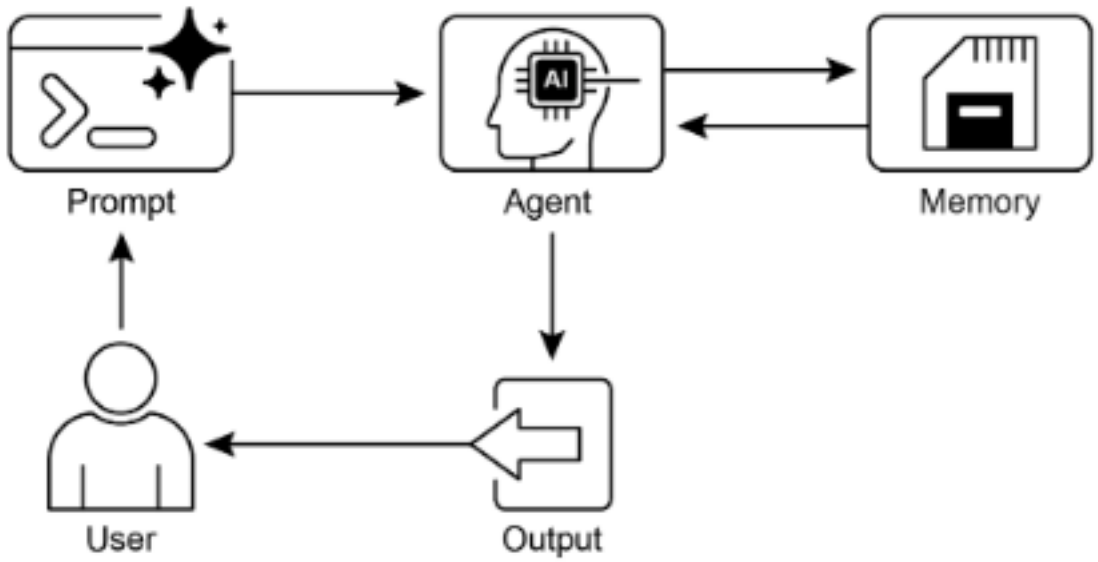

Figure 1: Memory Management Design Pattern

Fig.1: Memory management design pattern

圖 1：記憶管理設計模式

Key Takeaways

重點整理

To quickly recap the main points about memory management:

快速回顧記憶管理的重點：

- Memory is super important for agents to keep track of things, learn, and personalize interactions.

- Conversational AI relies on both short-term memory for immediate context within a single chat and long-term memory for persistent knowledge across multiple sessions.
- Short-term memory (the immediate stuff) is temporary, often limited by the LLM's context window or how the framework passes context.
- Long-term memory (the stuff that sticks around) saves info across different chats using outside storage like vector databases and is accessed by searching.
- Frameworks like ADK have specific parts like Session (the chat thread), State (temporary chat data), and MemoryService (the searchable long-term knowledge) to manage memory.
- ADK's SessionService handles the whole life of a chat session, including its history (events) and temporary data (state).
- ADK's session.state is a dictionary for temporary chat data. Prefixes (user:, app:, temp:) tell you where the data belongs and if it sticks around.
- In ADK, you should update state by using EventActions.state_delta or output_key when adding events, not by changing the state dictionary directly.
- ADK's MemoryService is for putting info into long-term storage and letting agents search it, often using tools.
- LangChain offers practical tools like ConversationBufferMemory to automatically inject the history of a single conversation into a prompt, enabling an agent to recall immediate context.
- LangGraph enables advanced, long-term memory by using a store to save and retrieve semantic facts, episodic experiences, or even updatable procedural rules across different user sessions.
- Memory Bank is a managed service that provides agents with persistent, long-term memory by automatically extracting, storing, and recalling user-specific information to enable personalized, continuous conversations across frameworks like Google's ADK, LangGraph, and CrewAI.
- 記憶對代理追蹤資訊、學習與個人化互動非常重要。
- 對話式 AI 同時依賴短期記憶（單次對話的即時脈絡）與長期記憶（跨多次 Session 的持久知識）。

- 短期記憶（即時資訊）是暫時的，常受限於 LLM 的上下文視窗或框架傳遞脈絡的方式。
- 長期記憶（能留存的資訊）會以外部儲存如向量資料庫跨對話保存，並透過搜尋存取。
- 像 ADK 這樣的框架有 Session（聊天執行緒）、State（暫存對話資料）與 MemoryService（可搜尋的長期知識）來管理記憶。
- ADK 的 SessionService 管理整個聊天 Session 的生命週期，包括歷史（events）與暫存資料（state）。
- ADK 的 session.state 是暫存對話資料的字典。前綴（user:、app:、temp:）說明資料歸屬與是否持久化。
- 在 ADK 中，更新 state 應使用 EventActions.state_delta 或 output_key 來新增事件，而不是直接改 state 字典。
- ADK 的 MemoryService 用於將資訊寫入長期儲存並讓代理搜尋，常搭配工具使用。
- LangChain 提供 ConversationBufferMemory 等實用工具，自動把單次對話歷史注入提示，讓代理回想即時脈絡。
- LangGraph 透過 store 保存與取回語意事實、情節經驗，或可更新的程序規則，實現進階長期記憶。
- Memory Bank 是受管理服務，透過自動抽取、儲存與回想使用者資訊，提供持久長期記憶，讓 Google ADK、LangGraph 與 CrewAI 等框架能進行個人化且連續的對話。

Conclusion

結論

This chapter dove into the really important job of memory management for agent systems, showing the difference between the short-lived context and the knowledge that sticks around for a long time. We talked about how these types of memory are set up and where you see them used in building smarter agents that can remember things. We took a detailed look at how Google ADK gives you specific pieces like Session, State, and MemoryService to handle this. Now that we've covered how agents can remember things, both short-term and long-term, we can move on to how they can learn and adapt. The next pattern

“Learning and Adaptation” is about an agent changing how it thinks, acts, or what it knows, all based on new experiences or data.

本章深入探討代理系統中記憶管理這項非常重要的工作，說明短暫脈絡與可長期保留知識之間的差異。我們談到這些記憶類型如何建立，以及在打造能記住資訊的智慧代理時如何應用。也詳細介紹了 Google ADK 透過 Session、State 與 MemoryService 等元件來處理記憶。既然我們已了解代理如何擁有短期與長期記憶，接下來就要談它們如何學習與適應。下一個模式「學習與適應」著重於代理如何根據新經驗或資料改變其思考方式、行為或知識。

References

參考資料

1. ADK Memory, https://google.github.io/adk-docs/sessions/memory/
2. LangGraph Memory, https://langchain-ai.github.io/langgraph/concepts/memory/
3. Vertex AI Agent Engine Memory Bank, https://cloud.google.com/blog/products/ai-machine-learning/vertex-ai-memory-bank-in-public-preview
4. ADK 記憶體， https://google.github.io/adk-docs/sessions/memory/
5. LangGraph 記憶體， https://langchain-ai.github.io/langgraph/concepts/memory/
6. Vertex AI Agent Engine Memory Bank， https://cloud.google.com/blog/products/ai-machine-learning/vertex-ai-memory-bank-in-public-preview

Chapter 9: Learning and Adaptation

第 9 章：學習與適應

Learning and adaptation are pivotal for enhancing the capabilities of artificial intelligence agents. These processes enable agents to evolve beyond predefined parameters, allowing them to improve autonomously through experience and environmental interaction. By learning and adapting, agents can effectively manage novel situations and optimize their performance without constant manual intervention. This chapter explores the principles and mechanisms underpinning agent learning and adaptation in detail.

學習與適應對提升人工智慧代理的能力至關重要。這些過程讓代理能超越預先定義的參數，透過經驗與環境互動自主改進。藉由學習與適應，代理能有效處理新情境並在

無需持續人工介入的情況下最佳化表現。本章將詳盡探討代理學習與適應的原理與機制。

The Big Picture

整體概觀

Agents learn and adapt by changing their thinking, actions, or knowledge based on new experiences and data. This allows agents to evolve from simply following instructions to becoming smarter over time.

代理透過根據新經驗與資料改變其思考、行動或知識來學習與適應。這使代理能從單純遵循指令，逐步演進為更聰明的系統。

- **Reinforcement Learning:** Agents try actions and receive rewards for positive outcomes and penalties for negative ones, learning optimal behaviors in changing situations. Useful for agents controlling robots or playing games.
- **Supervised Learning:** Agents learn from labeled examples, connecting inputs to desired outputs, enabling tasks like decision-making and pattern recognition. Ideal for agents sorting emails or predicting trends.
- **Unsupervised Learning:** Agents discover hidden connections and patterns in unlabeled data, aiding in insights, organization, and creating a mental map of their environment. Useful for agents exploring data without specific guidance.
- **Few-Shot/Zero-Shot Learning with LLM-Based Agents:** Agents leveraging LLMs can quickly adapt to new tasks with minimal examples or clear instructions, enabling rapid responses to new commands or situations.
- **Online Learning:** Agents continuously update knowledge with new data, essential for real-time reactions and ongoing adaptation in dynamic environments. Critical for agents processing continuous data streams.
- **Memory-Based Learning:** Agents recall past experiences to adjust current actions in similar situations, enhancing context awareness and decision-making. Effective for agents with memory recall capabilities.
- **強化學習：** 代理嘗試行動，對正向結果獲得獎勵、對負向結果受到懲罰，從而在變動情境中學到最佳行為。適合控制機器人或玩遊戲的代理。
- **監督式學習：** 代理從有標註的範例學習，把輸入對應到期望輸出，以進行決策或模式辨識等任務。適合做郵件分類或趨勢預測的代理。

- **非監督式學習：** 代理在無標註資料中發現隱藏關聯與模式，用於產生洞察、組織資料並建立環境的心智地圖。適合在無明確指引下探索資料的代理。
- **基於 LLM 的少量 / 零樣本學習：** 使用 LLM 的代理可在極少示例或清晰指令下快速適應新任務，迅速回應新命令或情境。
- **線上學習：** 代理持續用新資料更新知識，對即時反應與動態環境中的持續適應至關重要。適合處理連續資料流的代理。
- **記憶式學習：** 代理回想過去經驗，在類似情況下調整當前行動，提升情境感知與決策能力。適合具記憶回想能力的代理。

Agents adapt by changing strategy, understanding, or goals based on learning. This is vital for agents in unpredictable, changing, or new environments.

代理透過學習改變策略、理解或目標而適應。這對處於不可預測、變動或全新環境中的代理至關重要。

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm used to train agents in environments with a continuous range of actions, like controlling a robot's joints or a character in a game. Its main goal is to reliably and stably improve an agent's decision-making strategy, known as its policy.

近端策略最佳化（PPO） 是一種強化學習演算法，用於訓練具有連續動作空間的代理，例如控制機器人的關節或遊戲角色。它的主要目標是可靠且穩定地改進代理的決策策略（policy）。

The core idea behind PPO is to make small, careful updates to the agent's policy. It avoids drastic changes that could cause performance to collapse. Here's how it works:

PPO 的核心理念是對代理策略做小而謹慎的更新，避免劇烈變動導致效能崩潰。流程如下：

1. Collect Data: The agent interacts with its environment (e.g., plays a game) using its current policy and collects a batch of experiences (state, action, reward).
2. Evaluate a "Surrogate" Goal: PPO calculates how a potential policy update would change the expected reward. However, instead of just maximizing this reward, it uses a special "clipped" objective function.
3. The "Clipping" Mechanism: This is the key to PPO's stability. It creates a "trust region" or a safe zone around the current policy. The algorithm is

prevented from making an update that is too different from the current strategy. This clipping acts like a safety brake, ensuring the agent doesn't take a huge, risky step that undoes its learning.

4. 收集資料：代理用目前策略與環境互動（例如玩遊戲），蒐集一批經驗（狀態、動作、獎勵）。
5. 評估「替代」目標：PPO 計算潛在策略更新如何改變期望獎勵。然而它不只是最大化獎勵，而是使用特殊的「剪裁」目標函數。
6. 「剪裁」機制：這是 PPO 穩定性的關鍵。它在現有策略周圍建立一個「信任區」或安全區域，防止演算法做出與當前策略差異過大的更新。剪裁就像安全剎車，確保代理不會踏出過大、危險的一步而抹去已學成果。

In short, PPO balances improving performance with staying close to a known, working strategy, which prevents catastrophic failures during training and leads to more stable learning.

簡言之，PPO 在提升效能與維持可行策略的穩定性之間取得平衡，避免訓練中出現災難性失敗，並帶來更穩定的學習。

Direct Preference Optimization (DPO) is a more recent method designed specifically for aligning Large Language Models (LLMs) with human preferences. It offers a simpler, more direct alternative to using PPO for this task.

直接偏好最佳化（DPO） 是較新的方法，專為讓大型語言模型（LLM）對齊人類偏好而設計。它提供比使用 PPO 更簡單、直接的替代方案。

To understand DPO, it helps to first understand the traditional PPO-based alignment method:

要理解 DPO，先了解傳統的 PPO 對齊方法會更有幫助：

- The PPO Approach (Two-Step Process):
  1. Train a Reward Model: First, you collect human feedback data where people rate or compare different LLM responses (e.g., "Response A is better than Response B"). This data is used to train a separate AI model, called a reward model, whose job is to predict what score a human would give to any new response.
  2. Fine-Tune with PPO: Next, the LLM is fine-tuned using PPO. The LLM's goal is to generate responses that get the highest possible score from the reward model. The reward model acts as the "judge" in the training game.

- PPO 方法（兩步驟流程）：
  1. 訓練獎勵模型：先蒐集人類回饋資料，讓人比較或評分不同 LLM 回覆（例如「回覆 A 比回覆 B 好」）。這些資料用來訓練一個獨立 AI 模型，稱為獎勵模型，其任務是預測人類會給新回覆什麼分數。
  2. 以 PPO 微調：接著用 PPO 微調 LLM。LLM 的目標是產生可在獎勵模型中拿到最高分的回覆。獎勵模型在這個訓練遊戲中扮演「裁判」。

This two-step process can be complex and unstable. For instance, the LLM might find a loophole and learn to "hack" the reward model to get high scores for bad responses.

此兩步驟流程可能複雜且不穩定。例如 LLM 可能找出漏洞，學會「駭入」獎勵模型，對不良回覆也拿到高分。

- The DPO Approach (Direct Process): DPO skips the reward model entirely. Instead of translating human preferences into a reward score and then optimizing for that score, DPO uses the preference data directly to update the LLM's policy.
- It works by using a mathematical relationship that directly links preference data to the optimal policy. It essentially teaches the model: "Increase the probability of generating responses like the preferred one and decrease the probability of generating ones like the disfavored one."
- DPO 方法（直接流程）：DPO 完全跳過獎勵模型。它不將人類偏好轉成獎勵分數再去最佳化，而是直接使用偏好資料來更新 LLM 的策略。
- 它利用一種數學關係，直接把偏好資料連結到最佳策略。本質上是在教模型：「提高產生偏好回覆的機率，降低產生不偏好回覆的機率。」

In essence, DPO simplifies alignment by directly optimizing the language model on human preference data. This avoids the complexity and potential instability of training and using a separate reward model, making the alignment process more efficient and robust.

總之，DPO 透過直接用人類偏好資料來最佳化語言模型，簡化了對齊流程。這避免了訓練與使用獎勵模型的複雜性與潛在不穩定性，使對齊過程更有效且更穩健。

Practical Applications & Use Cases

實務應用與使用情境

Adaptive agents exhibit enhanced performance in variable environments through iterative updates driven by experiential data.

具適應性的代理會在變動環境中透過經驗資料驅動的迭代更新，展現更佳表現。

- **Personalized assistant agents** refine interaction protocols through longitudinal analysis of individual user behaviors, ensuring highly optimized response generation.
- **Trading bot agents** optimize decision–making algorithms by dynamically adjusting model parameters based on high–resolution, real–time market data, thereby maximizing financial returns and mitigating risk factors.
- **Application agents** optimize user interface and functionality through dynamic modification based on observed user behavior, resulting in increased user engagement and system intuitiveness.
- **Robotic and autonomous vehicle agents** enhance navigation and response capabilities by integrating sensor data and historical action analysis, enabling safe and efficient operation across diverse environmental conditions.
- **Fraud detection agents** improve anomaly detection by refining predictive models with newly identified fraudulent patterns, enhancing system security and minimizing financial losses.
- **Recommendation agents** improve content selection precision by employing user preference learning algorithms, providing highly individualized and contextually relevant recommendations.
- **Game AI agents** enhance player engagement by dynamically adapting strategic algorithms, thereby increasing game complexity and challenge.
- **Knowledge Base Learning Agents**: Agents can leverage Retrieval Augmented Generation (RAG) to maintain a dynamic knowledge base of problem descriptions and proven solutions (see the Chapter 14). By storing successful strategies and challenges encountered, the agent can reference this data during decision–making, enabling it to adapt to new situations more effectively by applying previously successful patterns or avoiding known pitfalls.

- **個人化助理代理** 透過長期分析個別使用者行為精煉互動流程，確保回應生成高度最佳化。
- **交易機器人代理** 依高解析度的即時市場資料動態調整模型參數，最佳化決策演算法，以最大化報酬並降低風險。
- **應用程式代理** 根據觀察到的使用者行為動態修改介面與功能，提升使用者參與度與系統直覺性。
- **機器人與自駕車代理** 結合感測器資料與歷史行動分析強化導航與回應能力，使其在多樣環境中安全且高效運作。
- **詐欺偵測代理** 透過新識別的詐欺模式精煉預測模型，提升系統安全並減少財務損失。
- **推薦代理** 透過使用者偏好學習演算法提高內容選擇精準度，提供高度個人化且具情境相關的推薦。
- **遊戲 AI 代理** 動態調整策略演算法以提升玩家參與度，進而增加遊戲複雜性與挑戰性。
- **知識庫學習代理：** 代理可利用檢索增強生成（RAG）維持動態知識庫，儲存問題描述與已證實解法（見第 14 章）。透過保存成功策略與遭遇的挑戰，代理在決策時可引用這些資料，藉由套用成功模式或避開已知陷阱，更有效地適應新情境。

Case Study: The Self-Improving Coding Agent (SICA)

案例研究：自我改進的程式碼代理（SICA）

The Self-Improving Coding Agent (SICA), developed by Maxime Robeyns, Laurence Aitchison, and Martin Szummer, represents an advancement in agent-based learning, demonstrating the capacity for an agent to modify its own source code. This contrasts with traditional approaches where one agent might train another; SICA acts as both the modifier and the modified entity, iteratively refining its code base to improve performance across various coding challenges.

由 Maxime Robeyns、Laurence Aitchison 與 Martin Szummer 開發的自我改進程式碼代理（SICA），代表了代理式學習的進展，展示代理修改自身原始碼的能力。這與傳統做法（由一個代理訓練另一個代理）不同；SICA 同時是修改者與被修改者，透過反覆迭代精煉其程式碼庫，以提升在各種程式挑戰中的表現。

SICA's self-improvement operates through an iterative cycle (see Fig.1). Initially, SICA reviews an archive of its past versions and their performance on

benchmark tests. It selects the version with the highest performance score, calculated based on a weighted formula considering success, time, and computational cost. This selected version then undertakes the next round of self–modification. It analyzes the archive to identify potential improvements and then directly alters its codebase. The modified agent is subsequently tested against benchmarks, with the results recorded in the archive. This process repeats, facilitating learning directly from past performance. This self–improvement mechanism allows SICA to evolve its capabilities without requiring traditional training paradigms.

SICA 的自我改進以迭代循環進行（見圖 1）。起初，SICA 會檢視過往版本的存檔及其在基準測試中的表現，並依據成功率、時間與計算成本的加權公式挑選表現最高的版本。被選中的版本接著進行下一輪自我修改：分析存檔以找出可能改進點，並直接修改其程式碼庫。修改後的代理再接受基準測試，結果記錄回存檔中。此流程不斷重複，讓它能直接從過去表現中學習。這種自我改進機制使 SICA 能在不依賴傳統訓練範式的情況下演進能力。

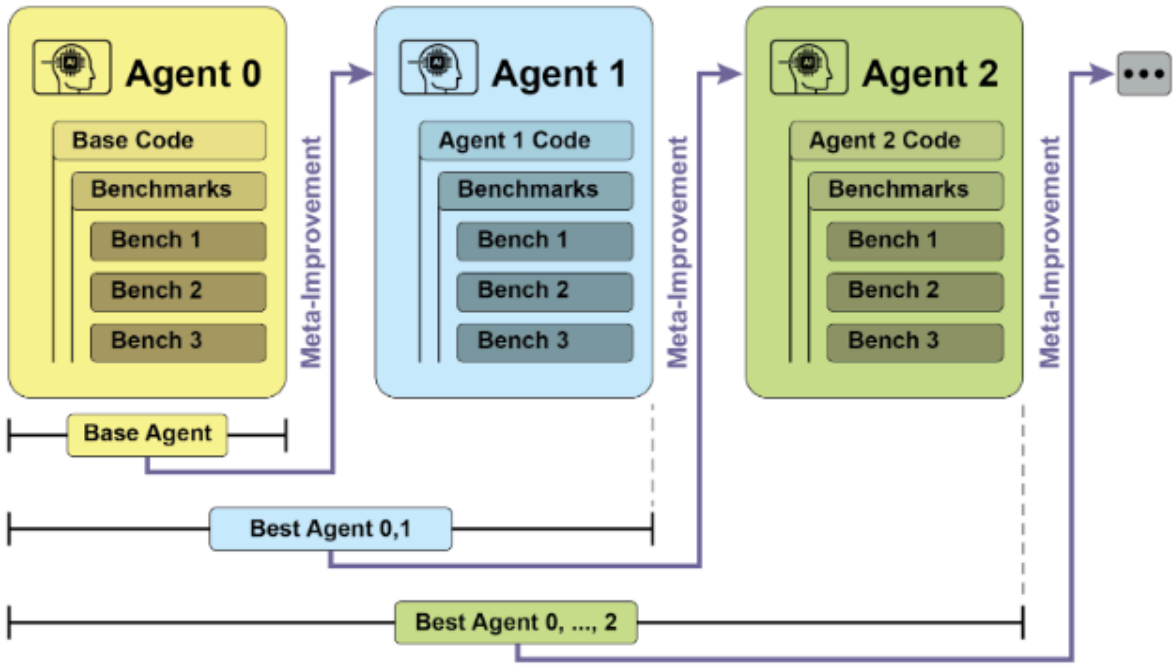

Figure 2: SICA's self–improvement, learning and adapting based on its past versions

Fig.1: SICA's self–improvement, learning and adapting based on its past versions

圖 1：SICA 的自我改進，根據過往版本進行學習與適應

SICA underwent significant self-improvement, leading to advancements in code editing and navigation. Initially, SICA utilized a basic file-overwriting approach for code changes. It subsequently developed a “Smart Editor” capable of more intelligent and contextual edits. This evolved into a “Diff-Enhanced Smart Editor,” incorporating diffs for targeted modifications and pattern-based editing, and a “Quick Overwrite Tool” to reduce processing demands.

SICA 經歷顯著的自我改進，帶來程式碼編輯與導航方面的進展。一開始，SICA 使用基本的檔案覆寫方式修改程式碼，之後發展出能進行更智慧且具脈絡編輯的「Smart Editor」。這進一步演化為「Diff-Enhanced Smart Editor」，透過 diff 進行目標式修改與樣式化編輯，並加入「Quick Overwrite Tool」以降低處理成本。

SICA further implemented “Minimal Diff Output Optimization” and “Context-Sensitive Diff Minimization,” using Abstract Syntax Tree (AST) parsing for efficiency. Additionally, a “SmartEditor Input Normalizer” was added. In terms of navigation, SICA independently created an “AST Symbol Locator,” using the code’s structural map (AST) to identify definitions within the codebase. Later, a “Hybrid Symbol Locator” was developed, combining a quick search with AST checking. This was further optimized via “Optimized AST Parsing in Hybrid Symbol Locator” to focus on relevant code sections, improving search speed. (see Fig. 2)

SICA 進一步實作「Minimal Diff Output Optimization」與「Context-Sensitive Diff Minimization」，利用抽象語法樹（AST）解析提升效率。此外，也加入了「SmartEditor Input Normalizer」。在導航方面，SICA 自行建立「AST Symbol Locator」，使用程式碼的結構地圖（AST）在程式庫中定位定義。之後又發展出結合快速搜尋與 AST 檢查的「Hybrid Symbol Locator」，並透過「Optimized AST Parsing in Hybrid Symbol Locator」進一步最佳化，使其更聚焦相關程式區段、提升搜尋速度。(見圖 2)

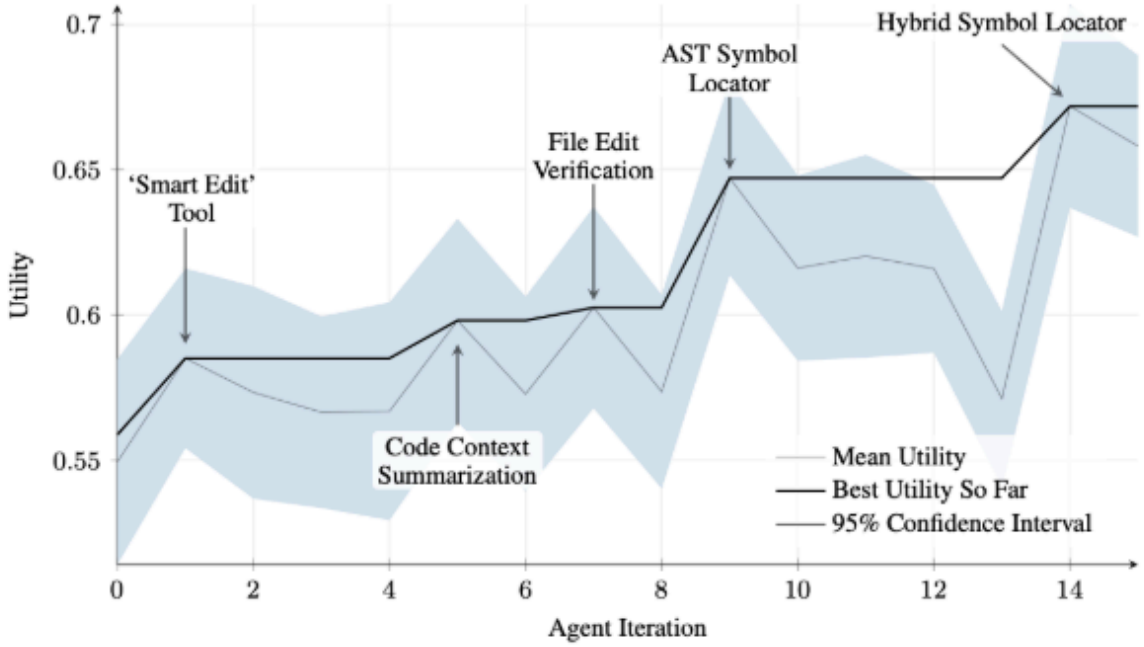

Figure 3: Performance across Iterations

Fig.2 : Performance across iterations. Key improvements are annotated with their corresponding tool or agent modifications. (courtesy of Maxime Robeyns , Martin Szummer , Laurence Aitchison)

圖 2：各次迭代的效能表現。關鍵改進以其對應的工具或代理修改標示。(圖片來源：Maxime Robeyns、Martin Szummer、Laurence Aitchison)

SICA's architecture comprises a foundational toolkit for basic file operations, command execution, and arithmetic calculations. It includes mechanisms for result submission and the invocation of specialized sub-agents (coding, problem-solving, and reasoning). These sub-agents decompose complex tasks and manage the LLM's context length, especially during extended improvement cycles.

SICA 的架構包含基礎工具組，用於基本檔案操作、指令執行與算術計算。它也包含結果提交機制與專門子代理（編碼、解題與推理）的呼叫。這些子代理負責拆解複雜任務並管理 LLM 的上下文長度，尤其是在延伸的改進循環中。

An asynchronous overseer, another LLM, monitors SICA's behavior, identifying potential issues such as loops or stagnation. It communicates with SICA and can intervene to halt execution if necessary. The overseer receives a detailed report of SICA's actions, including a callgraph and a log of messages and tool actions, to identify patterns and inefficiencies.

一個非同步監督者（另一個 LLM）負責監控 SICA 的行為，辨識如迴圈或停滯等問題。它會與 SICA 溝通，必要時可介入停止執行。監督者會收到 SICA 行為的詳細報告，包括呼叫圖與訊息、工具行為紀錄，用以辨識模式與低效率。

SICA's LLM organizes information within its context window, its short-term memory, in a structured manner crucial to its operation. This structure includes a System Prompt defining agent goals, tool and sub-agent documentation, and system instructions. A Core Prompt contains the problem statement or instruction, content of open files, and a directory map. Assistant Messages record the agent's step-by-step reasoning, tool and sub-agent call records and results, and overseer communications. This organization facilitates efficient information flow, enhancing LLM operation and reducing processing time and costs. Initially, file changes were recorded as diffs, showing only modifications and periodically consolidated.

SICA 的 LLM 會在其上下文視窗（短期記憶）中以結構化方式組織資訊，這對其運作至關重要。此結構包含定義代理目標的 System Prompt、工具與子代理文件，以及系統指令。Core Prompt 包含問題敘述或指令、已開啟檔案內容與目錄地圖。Assistant Messages 記錄代理的逐步推理、工具與子代理呼叫紀錄與結果，以及監督者的通訊。這種組織方式促進資訊高效流動，提升 LLM 的運作效率並降低時間與成本。起初，檔案變更以 diff 形式記錄，只顯示修改並定期整併。

SICA: A Look at the Code: Delving deeper into SICA's implementation reveals several key design choices that underpin its capabilities. As discussed, the system is built with a modular architecture, incorporating several sub-agents, such as a coding agent, a problem-solver agent, and a reasoning agent. These sub-agents are invoked by the main agent, much like tool calls, serving to decompose complex tasks and efficiently manage context length, especially during those extended meta-improvement iterations.

SICA：深入程式碼： 進一步探究 SICA 的實作可看到支撐其能力的多項關鍵設計選擇。如前所述，系統採模組化架構，包含多個子代理，例如編碼代理、解題代理與推理代理。這些子代理由主代理呼叫，形式類似工具呼叫，用於拆解複雜任務並有效管理上下文長度，特別是在延伸的自我改進迭代期間。

The project is actively developed and aims to provide a robust framework for those interested in post-training LLMs on tool use and other agentic tasks, with the full code available for further exploration and contribution at the https://github.com/MaximeRobeyns/self_improving_coding_agent/ GitHub repository.

該專案仍在積極開發中，旨在提供一個強健框架，供有興趣在工具使用與其他代理任務上進行 LLM 後訓練的人使用。完整程式碼可在 https://github.com/MaximeRobeyns/self_improving_coding_agent/ 的 GitHub 倉庫中進一步探索與貢獻。

For security, the project strongly emphasizes Docker containerization, meaning the agent runs within a dedicated Docker container. This is a crucial measure, as it provides isolation from the host machine, mitigating risks like inadvertent file system manipulation given the agent's ability to execute shell commands.

出於安全性考量，該專案強調使用 Docker 容器化，表示代理在專用 Docker 容器中執行。這是重要措施，因為它可與主機隔離，降低代理能執行 shell 指令所可能造成的檔案系統誤操作風險。

To ensure transparency and control, the system features robust observability through an interactive webpage that visualizes events on the event bus and the agent's callgraph. This offers comprehensive insights into the agent's actions, allowing users to inspect individual events, read overseer messages, and collapse sub-agent traces for clearer understanding.

為確保透明度與可控性，系統提供強大的可觀測性：透過互動式網頁視覺化事件匯流排上的事件與代理的呼叫圖。這提供對代理行為的全面洞察，讓使用者可檢視個別事件、閱讀監督者訊息，並折疊子代理軌跡以便更清楚理解。

In terms of its core intelligence, the agent framework supports LLM integration from various providers, enabling experimentation with different models to find the best fit for specific tasks. Finally, a critical component is the asynchronous overseer, an LLM that runs concurrently with the main agent. This overseer periodically assesses the agent's behavior for pathological deviations or stagnation and can intervene by sending notifications or even cancelling the agent's execution if necessary. It receives a detailed textual representation of the system's state, including a callgraph and an event stream of LLM messages, tool calls, and responses, which allows it to detect inefficient patterns or repeated work.

在核心智慧方面，代理框架支援多家供應商的 LLM 整合，使其能嘗試不同模型以找出最適合特定任務的選擇。最後，一個關鍵元件是非同步監督者（另一個 LLM），它與主代理並行運作。監督者會定期評估代理是否出現病態偏移或停滯，必要時可透過通知甚至取消執行進行介入。它會接收系統狀態的詳細文字表示，包括呼叫圖與 LLM 訊息、工具呼叫與回覆的事件流，以偵測低效率模式或重複作業。

A notable challenge in the initial SICA implementation was prompting the LLM-based agent to independently propose novel, innovative, feasible, and engaging modifications during each meta-improvement iteration. This limitation, particularly in fostering open-ended learning and authentic creativity in LLM agents, remains a key area of investigation in current research.

SICA 初期實作的一個顯著挑戰，是促使基於 LLM 的代理在每次自我改進迭代中，能獨立提出新穎、創新、可行且具吸引力的修改。這項限制，特別是在促進 LLM 代理開放式學習與真正創意方面，仍是目前研究的重要議題。

AlphaEvolve and OpenEvolve

AlphaEvolve 與 OpenEvolve

AlphaEvolve is an AI agent developed by Google designed to discover and optimize algorithms. It utilizes a combination of LLMs, specifically Gemini models (Flash and Pro), automated evaluation systems, and an evolutionary algorithm framework. This system aims to advance both theoretical mathematics and practical computing applications.

AlphaEvolve 是 Google 開發的 AI 代理，用於發現並最佳化演算法。它結合 LLM（特別是 Gemini 模型的 Flash 與 Pro 版本）、自動評估系統與演化演算法框架。此系統旨在推進理論數學與實務運算的應用。

AlphaEvolve employs an ensemble of Gemini models. Flash is used for generating a wide range of initial algorithm proposals, while Pro provides more in-depth analysis and refinement. Proposed algorithms are then automatically evaluated and scored based on predefined criteria. This evaluation provides feedback that is used to iteratively improve the solutions, leading to optimized and novel algorithms.

AlphaEvolve 使用 Gemini 模型的集成。Flash 用於產生大量初始演算法提案，而 Pro 提供更深入的分析與精煉。提出的演算法會依預先定義標準自動評估與打分，評估結果再用於迭代改進解法，產生最佳化且新穎的演算法。

In practical computing, AlphaEvolve has been deployed within Google's infrastructure. It has demonstrated improvements in data center scheduling, resulting in a 0.7% reduction in global compute resource usage. It has also contributed to hardware design by suggesting optimizations for Verilog code in upcoming Tensor Processing Units (TPUs). Furthermore, AlphaEvolve has accelerated AI performance, including a 23% speed improvement in a core

kernel of the Gemini architecture and up to 32.5% optimization of low-level GPU instructions for FlashAttention.

在實務運算中，AlphaEvolve 已部署於 Google 的基礎設施中。它在資料中心排程上帶來改善，使全球運算資源使用量降低 0.7%。它也透過為即將推出的 Tensor Processing Units（TPUs）之 Verilog 程式提出最佳化建議，貢獻硬體設計。此外，AlphaEvolve 也加速 AI 效能，包括 Gemini 架構核心內核提速 23%，以及將 FlashAttention 的低階 GPU 指令最佳化最多達 32.5%。

In the realm of fundamental research, AlphaEvolve has contributed to the discovery of new algorithms for matrix multiplication, including a method for 4x4 complex-valued matrices that uses 48 scalar multiplications, surpassing previously known solutions. In broader mathematical research, it has rediscovered existing state-of-the-art solutions to over 50 open problems in 75% of cases and improved upon existing solutions in 20% of cases, with examples including advancements in the kissing number problem.

在基礎研究領域，AlphaEvolve 促成了矩陣乘法新演算法的發現，例如對 4x4 複數矩陣使用 48 次純量乘法的方法，超越既有解法。在更廣泛的數學研究中，它在 75% 的案例中重新發現超過 50 個開放問題的既有最佳解，並在 20% 的案例中超越既有解法，包含親吻數問題的進展。

OpenEvolve is an evolutionary coding agent that leverages LLMs (see Fig.3) to iteratively optimize code. It orchestrates a pipeline of LLM-driven code generation, evaluation, and selection to continuously enhance programs for a wide range of tasks. A key aspect of OpenEvolve is its capability to evolve entire code files, rather than being limited to single functions. The agent is designed for versatility, offering support for multiple programming languages and compatibility with OpenAI-compatible APIs for any LLM. Furthermore, it incorporates multi-objective optimization, allows for flexible prompt engineering, and is capable of distributed evaluation to efficiently handle complex coding challenges.

OpenEvolve 是一個演化式程式碼代理，利用 LLM（見圖 3）迭代最佳化程式碼。它編排由 LLM 驅動的程式碼產生、評估與選擇流程，持續提升多種任務的程式。OpenEvolve 的關鍵特色是能演化整個程式檔，而不僅限於單一函式。該代理設計上具備高彈性，支援多種程式語言，並與任何符合 OpenAI API 的 LLM 相容。此外，它也納入多目標最佳化、允許彈性提示工程，並支援分散式評估以高效處理複雜程式挑戰。

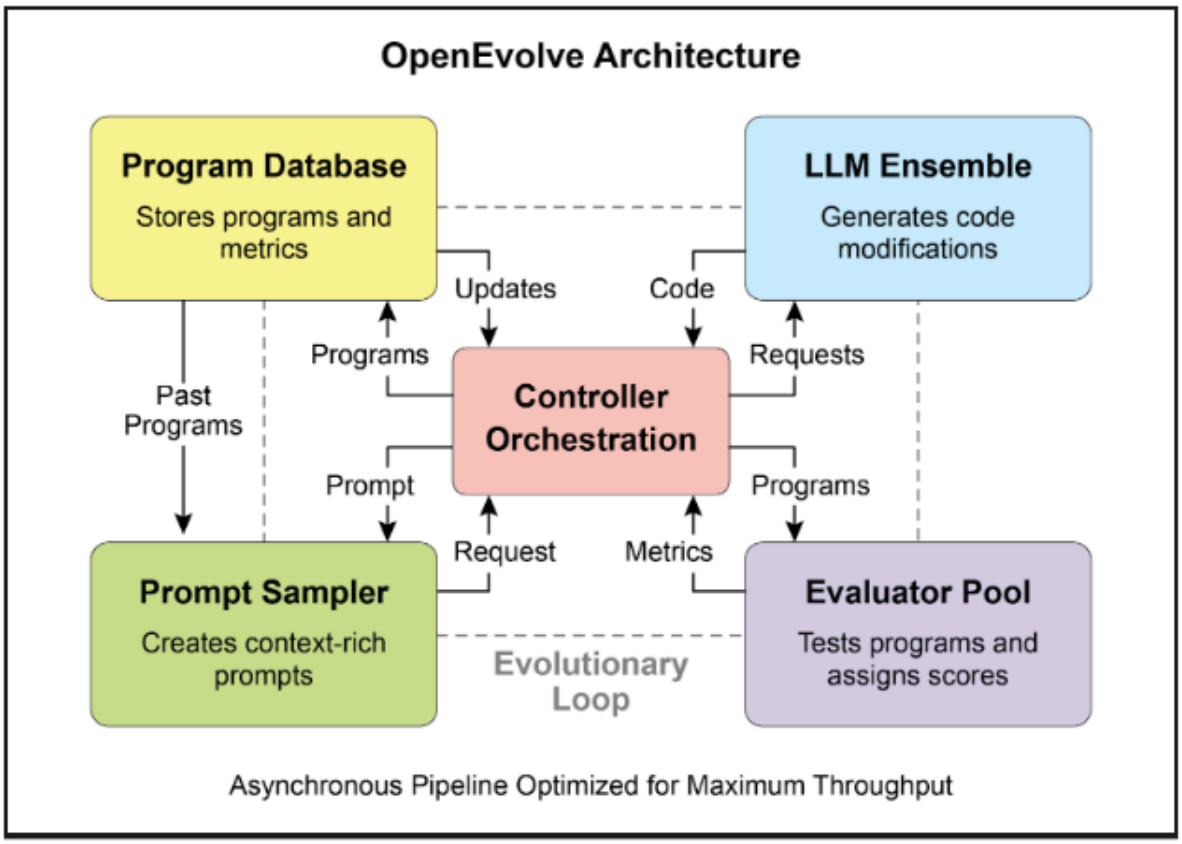

Figure 4: OpenEvolve Architecture

Fig. 3: The OpenEvolve internal architecture is managed by a controller. This controller orchestrates several key components: the program sampler, Program Database, Evaluator Pool, and LLM Ensembles. Its primary function is to facilitate their learning and adaptation processes to enhance code quality.

圖 3：OpenEvolve 內部架構由控制器管理。控制器協調多個關鍵元件：程式取樣器、程式資料庫、評估器池與 LLM 集合。其主要功能是促進學習與適應流程，以提升程式碼品質。

This code snippet uses the OpenEvolve library to perform evolutionary optimization on a program. It initializes the OpenEvolve system with paths to an initial program, an evaluation file, and a configuration file. The evolve.run(iterations=1000) line starts the evolutionary process, running for 1000 iterations to find an improved version of the program. Finally, it prints the metrics of the best program found during the evolution, formatted to four decimal places.

這段程式碼使用 OpenEvolve 函式庫對程式進行演化式最佳化。它以初始程式、評估檔與設定檔的路徑來初始化 OpenEvolve 系統。`evolve.run(iterations=1000)` 啟動演

化流程，執行 1000 次迭代以找到改進版本的程式。最後，它以四位小數格式輸出演化過程中找到的最佳程式之評估指標。

```
from openevolve import OpenEvolve


# Initialize the system
evolve = OpenEvolve(
    initial_program_path="path/to/initial_program.py",
    evaluation_file="path/to/evaluator.py",
    config_path="path/to/config.yaml",
)

# Run the evolution
best_program = await evolve.run(iterations=1000)

print("Best program metrics:")
for name, value in best_program.metrics.items():
    print(f"  {name}: {value:.4f}")
```

At a Glance

一覽

What: AI agents often operate in dynamic and unpredictable environments where pre-programmed logic is insufficient. Their performance can degrade when faced with novel situations not anticipated during their initial design. Without the ability to learn from experience, agents cannot optimize their strategies or personalize their interactions over time. This rigidity limits their effectiveness and prevents them from achieving true autonomy in complex, real-world scenarios.

什麼： AI 代理常在動態且不可預測的環境中運作，預先編寫的邏輯往往不足。當面對初始設計未預期的新情境時，代理的表現可能下降。若缺乏從經驗中學習的能力，代理就無法隨時間最佳化策略或個人化互動。這種僵化限制了其效能，也阻礙其在複雜真實場景中達成真正自主。

Why: The standardized solution is to integrate learning and adaptation mechanisms, transforming static agents into dynamic, evolving systems. This allows an agent to autonomously refine its knowledge and behaviors based on new data and interactions. Agentic systems can use various methods, from reinforcement learning to more advanced techniques like self-modification, as seen in the Self-Improving Coding Agent (SICA). Advanced systems like

Google's AlphaEvolve leverage LLMs and evolutionary algorithms to discover entirely new and more efficient solutions to complex problems. By continuously learning, agents can master new tasks, enhance their performance, and adapt to changing conditions without requiring constant manual reprogramming.

為什麼： 標準做法是導入學習與適應機制，把靜態代理轉成動態、可演進的系統。這讓代理能依新資料與互動自主精煉知識與行為。代理系統可採用從強化學習到自我修改等各種方法，如自我改進程式碼代理（SICA）所示。像 Google 的 AlphaEvolve 這類先進系統則運用 LLM 與演化演算法，發現全新且更高效的複雜問題解法。透過持續學習，代理可掌握新任務、提升效能並適應變化，而不需不斷人工重寫程式。

Rule of thumb: Use this pattern when building agents that must operate in dynamic, uncertain, or evolving environments. It is essential for applications requiring personalization, continuous performance improvement, and the ability to handle novel situations autonomously.

經驗法則： 當要打造必須在動態、不確定或持續演變環境中運作的代理時，就應使用此模式。對需要個人化、持續效能改進，以及能自主應對新情境的應用而言，這是必要模式。

Visual summary:

視覺摘要：

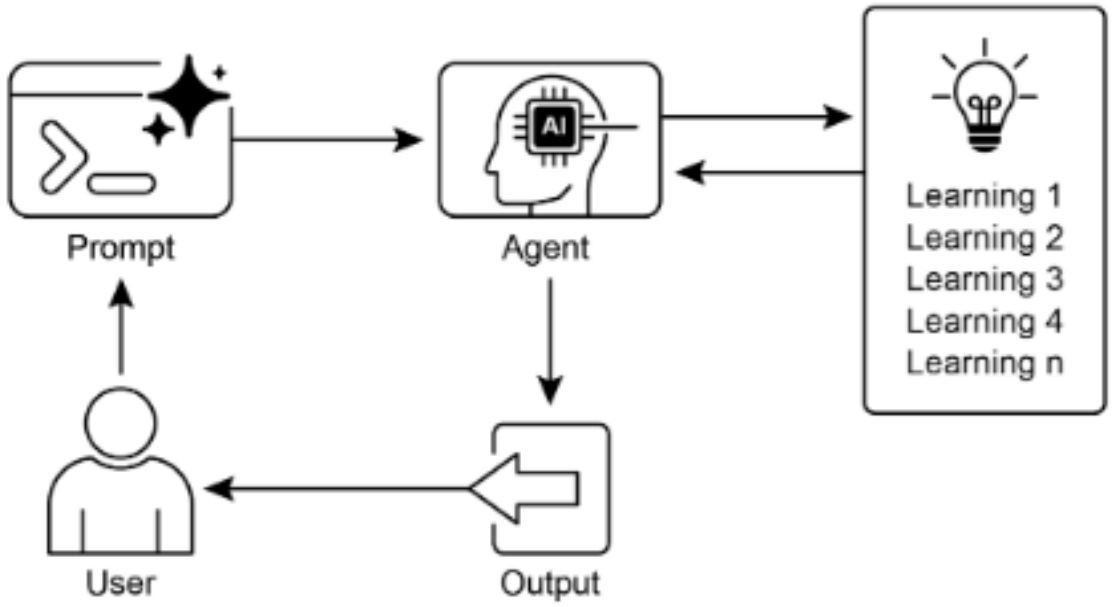

Figure 5: Learning and Adapting Pattern

Fig.4: Learning and adapting pattern

圖 4：學習與適應模式

Key Takeaways

重點整理

- Learning and Adaptation are about agents getting better at what they do and handling new situations by using their experiences.
- “Adaptation” is the visible change in an agent’s behavior or knowledge that comes from learning.
- SICA, the Self-Improving Coding Agent, self-improves by modifying its code based on past performance. This led to tools like the Smart Editor and AST Symbol Locator.
- Having specialized “sub-agents” and an “overseer” helps these self-improving systems manage big tasks and stay on track.
- The way an LLM’s “context window” is set up (with system prompts, core prompts, and assistant messages) is super important for how efficiently agents work.

- This pattern is vital for agents that need to operate in environments that are always changing, uncertain, or require a personal touch.
- Building agents that learn often means hooking them up with machine learning tools and managing how data flows.
- An agent system, equipped with basic coding tools, can autonomously edit itself, and thereby improve its performance on benchmark tasks
- AlphaEvolve is Google's AI agent that leverages LLMs and an evolutionary framework to autonomously discover and optimize algorithms, significantly enhancing both fundamental research and practical computing applications..
- 學習與適應是指代理利用經驗提升表現並處理新情境。
- 「適應」是源自學習的可見行為或知識變化。
- 自我改進程式碼代理 SICA 透過根據過去表現修改自身程式碼來自我改進，因而產生 Smart Editor 與 AST Symbol Locator 等工具。
- 具備專門子代理與「監督者」有助於這些自我改進系統管理大型任務並保持正軌。
- LLM 的「上下文視窗」如何設定（系統提示、核心提示與助理訊息）對代理效率極為重要。
- 此模式對需要在持續變動、不確定或需要個人化的環境中運作的代理至關重要。
- 打造具學習能力的代理通常需要串接機器學習工具並管理資料流。
- 配備基本程式工具的代理系統可自主編輯自身，進而提升基準任務的表現。
- AlphaEvolve 是 Google 的 AI 代理，利用 LLM 與演化框架自動發現並最佳化演算法，大幅推進基礎研究與實務運算應用。

Conclusion

結論

This chapter examines the crucial roles of learning and adaptation in Artificial Intelligence. AI agents enhance their performance through continuous data acquisition and experience. The Self-Improving Coding Agent (SICA) exemplifies this by autonomously improving its capabilities through code modifications.

本章探討學習與適應在人工智慧中的關鍵角色。AI 代理透過持續資料獲取與經驗累積來提升表現。自我改進程式碼代理（SICA）即是典型例子，能透過修改程式碼自主提升能力。

We have reviewed the fundamental components of agentic AI, including architecture, applications, planning, multi-agent collaboration, memory management, and learning and adaptation. Learning principles are particularly vital for coordinated improvement in multi-agent systems. To achieve this, tuning data must accurately reflect the complete interaction trajectory, capturing the individual inputs and outputs of each participating agent.

我們已回顧代理式 AI 的基本元件，包括架構、應用、規劃、多代理協作、記憶管理，以及學習與適應。學習原則對多代理系統的協同改進尤其重要。為達成這點，微調資料必須準確反映完整互動軌跡，涵蓋每個參與代理的輸入與輸出。

These elements contribute to significant advancements, such as Google's AlphaEvolve. This AI system independently discovers and refines algorithms by LLMs, automated assessment, and an evolutionary approach, driving progress in scientific research and computational techniques. Such patterns can be combined to construct sophisticated AI systems. Developments like AlphaEvolve demonstrate that autonomous algorithmic discovery and optimization by AI agents are attainable.

這些元素促成了如 Google 的 AlphaEvolve 等重大進展。此 AI 系統透過 LLM、自動評估與演化方法，獨立發現與精煉演算法，推動科學研究與計算技術的進步。這些模式可組合成更複雜的 AI 系統。AlphaEvolve 等發展顯示，AI 代理自主進行演算法發現與最佳化是可達成的。

References

參考資料

1. Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press.
2. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
3. Mitchell, T. M. (1997). Machine Learning. McGraw-Hill.
4. **Proximal Policy Optimization Algorithms** by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. You can find it on arXiv: https://arxiv.org/abs/1707.06347
5. Robeyns, M., Aitchison, L., & Szummer, M. (2025). A Self-Improving Coding Agent. arXiv:2504.15228v2. https://arxiv.org/pdf/2504.15228 https://github.com/MaximeRobeyns/self_improving_coding_agent

6. AlphaEvolve blog, https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/
7. OpenEvolve, https://github.com/codelion/openevolve
8. Sutton, R. S., & Barto, A. G.（2018）。Reinforcement Learning: An Introduction。MIT Press。
9. Goodfellow, I., Bengio, Y., & Courville, A.（2016）。Deep Learning。MIT Press。
10. Mitchell, T. M.（1997）。Machine Learning。McGraw-Hill。
11. **Proximal Policy Optimization Algorithms**（近端策略最佳化演算法），John Schulman、Filip Wolski、Prafulla Dhariwal、Alec Radford、Oleg Klimov。arXiv： https://arxiv.org/abs/1707.06347
12. Robeyns, M., Aitchison, L., & Szummer, M.（2025）。A Self-Improving Coding Agent。arXiv:2504.15228v2。 https://arxiv.org/pdf/2504.15228 https://github.com/MaximeRobeyns/self_improving_coding_agent
13. AlphaEvolve 部落格， https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/
14. OpenEvolve， https://github.com/codelion/openevolve

Chapter 10: Model Context Protocol

第 10 章：模型上下文協定（MCP）

To enable LLMs to function effectively as agents, their capabilities must extend beyond multimodal generation. Interaction with the external environment is necessary, including access to current data, utilization of external software, and execution of specific operational tasks. The Model Context Protocol (MCP) addresses this need by providing a standardized interface for LLMs to interface with external resources. This protocol serves as a key mechanism to facilitate consistent and predictable integration.

要讓 LLM 有效作為代理運作，其能力必須超越多模態生成。代理需要與外部環境互動，包括存取即時資料、使用外部軟體，以及執行特定操作任務。模型上下文協定（MCP）透過提供標準化介面來滿足此需求，使 LLM 能與外部資源對接。此協定是促進一致且可預測整合的重要機制。

MCP Pattern Overview

MCP 模式概覽

Imagine a universal adapter that allows any LLM to plug into any external system, database, or tool without a custom integration for each one. That's essentially what the Model Context Protocol (MCP) is. It's an open standard designed to standardize how LLMs like Gemini, OpenAI's GPT models, Mixtral, and Claude communicate with external applications, data sources, and tools. Think of it as a universal connection mechanism that simplifies how LLMs obtain context, execute actions, and interact with various systems.

想像有一個通用轉接器，讓任何 LLM 都能在不為每個系統客製整合的情況下，連接任何外部系統、資料庫或工具。這就是模型上下文協定（MCP）的概念。它是一個開放標準，旨在標準化 Gemini、OpenAI GPT、Mixtral、Claude 等 LLM 與外部應用、資料來源與工具的通訊方式。把它想成通用的連線機制，可簡化 LLM 取得脈絡、執行動作並與各系統互動的流程。

MCP operates on a client-server architecture. It defines how different elements—data (referred to as resources), interactive templates (which are essentially prompts), and actionable functions (known as tools)—are exposed by an MCP server. These are then consumed by an MCP client, which could be an LLM host application or an AI agent itself. This standardized approach dramatically reduces the complexity of integrating LLMs into diverse operational environments.

MCP 採用用戶端 / 伺服器架構。它定義 MCP 伺服器如何暴露不同元素——資料（稱為 resources）、互動式模板（本質上是 prompts）以及可執行功能（稱為 tools）。這些再由 MCP 用戶端（可能是 LLM 宿主應用或 AI 代理本身）使用。此標準化方法大幅降低 LLM 整合至多樣化營運環境的複雜度。

However, MCP is a contract for an "agentic interface," and its effectiveness depends heavily on the design of the underlying APIs it exposes. There is a risk that developers simply wrap pre-existing, legacy APIs without modification, which can be suboptimal for an agent. For example, if a ticketing system's API only allows retrieving full ticket details one by one, an agent asked to summarize high-priority tickets will be slow and inaccurate at high volumes. To be truly effective, the underlying API should be improved with deterministic features like filtering and sorting to help the non-deterministic agent work

efficiently. This highlights that agents do not magically replace deterministic workflows; they often require stronger deterministic support to succeed.

然而，MCP 是「代理介面」的契約，其效果高度依賴所暴露的底層 API 設計。若開發者只是把既有舊式 API 直接包起來，對代理而言可能並不理想。例如，如果工單系統 API 只能逐一取得完整工單，當代理要摘要高優先工單時，大量情況下將會緩慢且不準確。要真正有效，底層 API 應加入如篩選與排序等確定性功能，協助非確定性代理高效運作。這凸顯代理無法神奇取代確定性流程；反而常需要更強的確定性支援才能成功。

Furthermore, MCP can wrap an API whose input or output is still not inherently understandable by the agent. An API is only useful if its data format is agent-friendly, a guarantee that MCP itself does not enforce. For instance, creating an MCP server for a document store that returns files as PDFs is mostly useless if the consuming agent cannot parse PDF content. The better approach would be to first create an API that returns a textual version of the document, such as Markdown, which the agent can actually read and process. This demonstrates that developers must consider not just the connection, but the nature of the data being exchanged to ensure true compatibility.

此外，MCP 也可能包裝輸入或輸出對代理仍不易理解的 API。API 只有在資料格式對代理友善時才有用，而 MCP 本身不保證這點。例如，若為文件儲存建立一個回傳 PDF 的 MCP 伺服器，對無法解析 PDF 的代理幾乎沒有價值。更好的作法是先提供可回傳文字版（如 Markdown）的 API，讓代理能讀取與處理。這說明開發者需考量的不只是連線，還有交換資料的性質，以確保真正相容。

MCP vs. Tool Function Calling

MCP 與工具函式呼叫的比較

The Model Context Protocol (MCP) and tool function calling are distinct mechanisms that enable LLMs to interact with external capabilities (including tools) and execute actions. While both serve to extend LLM capabilities beyond text generation, they differ in their approach and level of abstraction.

模型上下文協定（MCP）與工具函式呼叫是兩種讓 LLM 互動外部能力（含工具）並執行動作的不同機制。雖然兩者都能擴展 LLM 的能力，超越文字生成，但在方法與抽象層級上有所不同。

Tool function calling can be thought of as a direct request from an LLM to a specific, pre-defined tool or function. Note that in this context we use the

words “tool” and “function” interchangeably. This interaction is characterized by a one-to-one communication model, where the LLM formats a request based on its understanding of a user’s intent requiring external action. The application code then executes this request and returns the result to the LLM. This process is often proprietary and varies across different LLM providers.

工具函式呼叫可視為 LLM 對特定、預先定義工具或函式的直接請求。在此情境中「tool」與「function」可互換。此互動採一對一通訊模型，LLM 依對使用者意圖的理解來格式化請求，應用程式再執行該請求並回傳結果給 LLM。此流程常為專有實作，且依不同 LLM 供應商而異。

In contrast, the Model Context Protocol (MCP) operates as a standardized interface for LLMs to discover, communicate with, and utilize external capabilities. It functions as an open protocol that facilitates interaction with a wide range of tools and systems, aiming to establish an ecosystem where any compliant tool can be accessed by any compliant LLM. This fosters interoperability, composability and reusability across different systems and implementations. By adopting a federated model, we significantly improve interoperability and unlock the value of existing assets. This strategy allows us to bring disparate and legacy services into a modern ecosystem simply by wrapping them in an MCP-compliant interface. These services continue to operate independently, but can now be composed into new applications and workflows, with their collaboration orchestrated by LLMs. This fosters agility and reusability without requiring costly rewrites of foundational systems.

相較之下，MCP 作為標準化介面，讓 LLM 能發現、溝通與使用外部能力。它是開放協定，促成 LLM 與廣泛工具與系統互動，目標是建立任何相容工具都能被任何相容 LLM 存取的生態系。這促進不同系統與實作間的互通性、可組合性與可重用性。採用聯邦式模型可顯著提升互通性並釋放既有資產價值。此策略讓我們能透過 MCP 相容介面包裝，將異質與舊式服務帶入現代生態系。這些服務仍獨立運作，但如今可被組合成新的應用與流程，並由 LLM 編排協作，提升敏捷性與重用性而無須昂貴的重寫。

Here’s a breakdown of the fundamental distinctions between MCP and tool function calling:

以下為 MCP 與工具函式呼叫的基本差異：

Feature	Tool Function Calling	Model Context Protocol (MCP)
Standardization	Proprietary and vendor-specific. The format and implementation differ across LLM providers.	An open, standardized protocol, promoting interoperability between different LLMs and tools.
Scope	A direct mechanism for an LLM to request the execution of a specific, predefined function.	A broader framework for how LLMs and external tools discover and communicate with each other.
Architecture	A one-to-one interaction between the LLM and the application's tool-handling logic.	A client-server architecture where LLM-powered applications (clients) can connect to and utilize various MCP servers (tools).
Discovery	The LLM is explicitly told which tools are available within the context of a specific conversation.	Enables dynamic discovery of available tools. An MCP client can query a server to see what capabilities it offers.
Reusability	Tool integrations are often tightly coupled with the specific application and LLM being used.	Promotes the development of reusable, standalone "MCP servers" that can be accessed by any compliant application.

功能	工具函式呼叫	模型上下文協定（MCP）
標準化	專有且依廠商而異，格式與實作在不同 LLM 供應商間不同。	開放的標準化協定，促進不同 LLM 與工具間的互通性。
範圍	LLM 直接請求執行特定、預先定義函式的機制。	LLM 與外部工具如何發現並相互通訊的更廣泛框架。
架構	LLM 與應用程式的工具處理邏輯之間的一對一互動。	用戶端 / 伺服器架構，LLM 應用（用戶端）可連接並使用多個 MCP 伺服器（工具）。
發現能力	LLM 需明確被告知在特定對話中可用的工具。	允許動態發現可用工具，MCP 用戶端可查詢伺服器的能力。
可重用性	工具整合常與特定應用與 LLM 緊密綁定。	促進可重用、獨立的「MCP 伺服器」開發，可被任何相容應用存取。

Think of tool function calling as giving an AI a specific set of custom-built tools, like a particular wrench and screwdriver. This is efficient for a workshop with a fixed set of tasks. MCP (Model Context Protocol), on the other hand, is like creating a universal, standardized power outlet system. It doesn't provide the tools itself, but it allows any compliant tool from any manufacturer to plug in and work, enabling a dynamic and ever-expanding workshop.

可將工具函式呼叫想像成給 AI 一組特製工具，如特定扳手與螺絲起子。這對任務固定的工坊很有效率。而 MCP（模型上下文協定）則像建立通用、標準化的電源插座系統。它本身不提供工具，但允許任何相容廠商的工具插上就能使用，讓工坊動態且持續擴張。

In short, function calling provides direct access to a few specific functions, while MCP is the standardized communication framework that lets LLMs discover and use a vast range of external resources. For simple applications, specific tools are enough; for complex, interconnected AI systems that need to adapt, a universal standard like MCP is essential.

總結來說，函式呼叫提供對少數特定功能的直接存取，而 MCP 是標準化的通訊框架，讓 LLM 能發現並使用廣泛的外部資源。簡單應用使用特定工具即可；對需要適應的複雜互聯 AI 系統而言，像 MCP 這樣的通用標準則不可或缺。

Additional considerations for MCP

MCP 的其他考量

While MCP presents a powerful framework, a thorough evaluation requires considering several crucial aspects that influence its suitability for a given use case. Let's see some aspects in more details:

雖然 MCP 提供強大框架，但要評估其是否適合特定用例，仍需考量多項關鍵面向。以下是更詳細的說明：

- **Tool vs. Resource vs. Prompt**: It's important to understand the specific roles of these components. A resource is static data (e.g., a PDF file, a database record). A tool is an executable function that performs an action (e.g., sending an email, querying an API). A prompt is a template that guides the LLM in how to interact with a resource or tool, ensuring the interaction is structured and effective.
- **Discoverability**: A key advantage of MCP is that an MCP client can dynamically query a server to learn what tools and resources it offers. This "just-in-time" discovery mechanism is powerful for agents that need to adapt to new capabilities without being redeployed.
- **Security**: Exposing tools and data via any protocol requires robust security measures. An MCP implementation must include authentication and authorization to control which clients can access which servers and what specific actions they are permitted to perform.
- **Implementation**: While MCP is an open standard, its implementation can be complex. However, providers are beginning to simplify this process. For example, some model providers like Anthropic or FastMCP offer SDKs that abstract away much of the boilerplate code, making it easier for developers to create and connect MCP clients and servers.
- **Error Handling**: A comprehensive error-handling strategy is critical. The protocol must define how errors (e.g., tool execution failure, unavailable server, invalid request) are communicated back to the LLM so it can understand the failure and potentially try an alternative approach.

- **Local vs. Remote Server**: MCP servers can be deployed locally on the same machine as the agent or remotely on a different server. A local server might be chosen for speed and security with sensitive data, while a remote server architecture allows for shared, scalable access to common tools across an organization.
- **On-demand vs. Batch**: MCP can support both on-demand, interactive sessions and larger-scale batch processing. The choice depends on the application, from a real-time conversational agent needing immediate tool access to a data analysis pipeline that processes records in batches.
- **Transportation Mechanism**: The protocol also defines the underlying transport layers for communication. For local interactions, it uses JSON-RPC over STDIO (standard input/output) for efficient inter-process communication. For remote connections, it leverages web-friendly protocols like Streamable HTTP and Server-Sent Events (SSE) to enable persistent and efficient client-server communication.
- **工具 vs. 資源 vs. 提示：** 需要理解這些元件的角色。資源是靜態資料（如 PDF 檔或資料庫紀錄）。工具是可執行功能，用於完成動作（如寄信、呼叫 API）。提示是引導 LLM 與資源或工具互動的模板，確保互動有結構且有效。
- **可發現性：** MCP 的關鍵優勢之一是用戶端可動態查詢伺服器，以了解其提供的工具與資源。這種「即時發現」機制對需要適應新能力而不重新部署的代理相當有力。
- **安全性：** 透過任何協定暴露工具與資料都需要強健的安全措施。MCP 實作必須包含驗證與授權機制，以控制哪些用戶端可存取哪些伺服器與允許執行哪些動作。
- **實作：** 雖然 MCP 是開放標準，但實作可能複雜。不過供應商正簡化此流程，例如 Anthropic 或 FastMCP 提供的 SDK 可隱藏大量樣板碼，讓開發者更容易建立並連接 MCP 用戶端與伺服器。
- **錯誤處理：** 完整的錯誤處理策略很關鍵。協定必須定義錯誤（如工具執行失敗、伺服器不可用、請求無效）如何回傳給 LLM，使其能理解失敗並可能嘗試替代方案。
- **本地 vs. 遠端伺服器：** MCP 伺服器可部署在代理同機（本地）或不同伺服器（遠端）。本地伺服器可兼顧速度與敏感資料安全；遠端架構則能在組織內共享、擴展常用工具的存取。
- **即時 vs. 批次：** MCP 可支援即時互動與大規模批次處理，選擇取決於應用情境，從需要即時工具存取的對話代理到批次處理資料的分析管線皆可。

- **傳輸機制：** 協定也定義通訊底層傳輸層。本地互動使用 STDIO 上的 JSON-RPC 做高效率行程間通訊；遠端連線則使用如 Streamable HTTP 與 Server-Sent Events（SSE）等網路友善協定，以實現持久且高效的用戶端 / 伺服器通訊。

The Model Context Protocol uses a client-server model to standardize information flow. Understanding component interaction is key to MCP's advanced agentic behavior:

模型上下文協定使用用戶端 / 伺服器模型來標準化資訊流。理解元件間互動是 MCP 展現進階代理行為的關鍵：

1. **Large Language Model (LLM)**: The core intelligence. It processes user requests, formulates plans, and decides when it needs to access external information or perform an action.
2. **MCP Client**: This is an application or wrapper around the LLM. It acts as the intermediary, translating the LLM's intent into a formal request that conforms to the MCP standard. It is responsible for discovering, connecting to, and communicating with MCP Servers.
3. **MCP Server**: This is the gateway to the external world. It exposes a set of tools, resources, and prompts to any authorized MCP Client. Each server is typically responsible for a specific domain, such as a connection to a company's internal database, an email service, or a public API.
4. **Optional Third-Party (3P) Service:** This represents the actual external tool, application, or data source that the MCP Server manages and exposes. It is the ultimate endpoint that performs the requested action, such as querying a proprietary database, interacting with a SaaS platform, or calling a public weather API.
5. **大型語言模型（LLM）：** 核心智慧，處理使用者請求、規劃流程，並決定何時需要存取外部資訊或執行動作。
6. **MCP 用戶端：** 包裹 LLM 的應用或封裝層，作為中介，把 LLM 的意圖轉為符合 MCP 標準的正式請求，負責探索、連線並與 MCP 伺服器通訊。
7. **MCP 伺服器：** 對外世界的入口，向授權的 MCP 用戶端暴露工具、資源與提示。每個伺服器通常負責特定領域，如公司內部資料庫、電子郵件服務或公開 API。
8. **可選第三方（3P）服務：** 代表 MCP 伺服器所管理與暴露的實際外部工具、應用或資料來源，是真正執行請求的終端點，例如查詢專有資料庫、與 SaaS 平台互動或呼叫天氣 API。

The interaction flows as follows:

互動流程如下：

1. **Discovery**: The MCP Client, on behalf of the LLM, queries an MCP Server to ask what capabilities it offers. The server responds with a manifest listing its available tools (e.g., send_email), resources (e.g., customer_database), and prompts.
2. **Request Formulation**: The LLM determines that it needs to use one of the discovered tools. For instance, it decides to send an email. It formulates a request, specifying the tool to use (send_email) and the necessary parameters (recipient, subject, body).
3. **Client Communication**: The MCP Client takes the LLM's formulated request and sends it as a standardized call to the appropriate MCP Server.
4. **Server Execution**: The MCP Server receives the request. It authenticates the client, validates the request, and then executes the specified action by interfacing with the underlying software (e.g., calling the send() function of an email API).
5. **Response and Context Update**: After execution, the MCP Server sends a standardized response back to the MCP Client. This response indicates whether the action was successful and includes any relevant output (e.g., a confirmation ID for the sent email). The client then passes this result back to the LLM, updating its context and enabling it to proceed with the next step of its task.
6. **發現：** MCP 用戶端代表 LLM 向 MCP 伺服器查詢其能力。伺服器回傳清單，列出可用工具（如 send_email）、資源（如 customer_database）與提示。
7. **請求形成：** LLM 判斷需要使用某個已發現的工具。例如決定寄信，並形成請求，指定工具（send_email）與必要參數（收件者、主旨、內文）。
8. **用戶端通訊：** MCP 用戶端將 LLM 形成的請求，以標準化呼叫送至對應的 MCP 伺服器。
9. **伺服器執行：** MCP 伺服器接收請求，進行驗證與請求檢查，並透過底層軟體執行指定動作（如呼叫郵件 API 的 send() 函式）。

10. **回應與脈絡更新：** 執行後，MCP 伺服器回傳標準回應給 MCP 用戶端，說明動作是否成功並附上相關輸出（如寄信確認 ID）。用戶端再把結果傳回 LLM，更新其脈絡，使其得以進行下一步。

Practical Applications & Use Cases

實務應用與使用情境

MCP significantly broadens AI/LLM capabilities, making them more versatile and powerful. Here are nine key use cases:

MCP 大幅擴展 AI / LLM 的能力，使其更具彈性與威力。以下為九大關鍵使用情境：

- **Database Integration:** MCP allows LLMs and agents to seamlessly access and interact with structured data in databases. For instance, using the MCP Toolbox for Databases, an agent can query Google BigQuery datasets to retrieve real-time information, generate reports, or update records, all driven by natural language commands.
- **Generative Media Orchestration:** MCP enables agents to integrate with advanced generative media services. Through MCP Tools for Genmedia Services, an agent can orchestrate workflows involving Google's Imagen for image generation, Google's Veo for video creation, Google's Chirp 3 HD for realistic voices, or Google's Lyria for music composition, allowing for dynamic content creation within AI applications.
- **External API Interaction:** MCP provides a standardized way for LLMs to call and receive responses from any external API. This means an agent can fetch live weather data, pull stock prices, send emails, or interact with CRM systems, extending its capabilities far beyond its core language model.
- **Reasoning-Based Information Extraction:** Leveraging an LLM's strong reasoning skills, MCP facilitates effective, query-dependent information extraction that surpasses conventional search and retrieval systems. Instead of a traditional search tool returning an entire document, an agent can analyze the text and extract the precise clause, figure, or statement that directly answers a user's complex question.
- **Custom Tool Development:** Developers can build custom tools and expose them via an MCP server (e.g., using FastMCP). This allows specialized internal functions or proprietary systems to be made available to LLMs and other

agents in a standardized, easily consumable format, without needing to modify the LLM directly.

- **Standardized LLM-to-Application Communication:** MCP ensures a consistent communication layer between LLMs and the applications they interact with. This reduces integration overhead, promotes interoperability between different LLM providers and host applications, and simplifies the development of complex agentic systems.
- **Complex Workflow Orchestration:** By combining various MCP-exposed tools and data sources, agents can orchestrate highly complex, multi-step workflows. An agent could, for example, retrieve customer data from a database, generate a personalized marketing image, draft a tailored email, and then send it, all by interacting with different MCP services.
- **IoT Device Control:** MCP can facilitate LLM interaction with Internet of Things (IoT) devices. An agent could use MCP to send commands to smart home appliances, industrial sensors, or robotics, enabling natural language control and automation of physical systems.
- **Financial Services Automation:** In financial services, MCP could enable LLMs to interact with various financial data sources, trading platforms, or compliance systems. An agent might analyze market data, execute trades, generate personalized financial advice, or automate regulatory reporting, all while maintaining secure and standardized communication.
- **資料庫整合：** MCP 讓 LLM 與代理能無縫存取並互動於資料庫中的結構化資料。例如使用 MCP Toolbox for Databases，代理可查詢 Google BigQuery 資料集以取得即時資訊、產生報表或更新紀錄，全由自然語言指令驅動。
- **生成式媒體編排：** MCP 讓代理可整合先進的生成式媒體服務。透過 MCP Tools for Genmedia Services，代理可編排包含 Google Imagen 影像生成、Google Veo 影片生成、Google Chirp 3 HD 擬真語音，以及 Google Lyria 音樂創作的流程，讓 AI 應用內的內容生成更具動態性。
- **外部 API 互動：** MCP 提供 LLM 呼叫並接收任何外部 API 回應的標準方式。這代表代理能取得即時天氣、股價、寄送郵件或與 CRM 系統互動，大幅擴展其能力。
- **基於推理的資訊擷取：** 借助 LLM 強大的推理能力，MCP 能進行超越傳統搜尋與檢索系統的、依查詢而定的資訊擷取。代理可分析文本並抽取直接回答複雜問題的精確條款、數字或敘述，而非回傳整份文件。

- **自訂工具開發：** 開發者可建立自訂工具並透過 MCP 伺服器暴露（例如使用 FastMCP）。這讓專用內部功能或專有系統以標準且易用的格式提供給 LLM 與其他代理，而無需修改 LLM 本體。
- **標準化 LLM 與應用溝通：** MCP 確保 LLM 與其互動的應用之間有一致的通訊層，降低整合成本，促進不同 LLM 供應商與宿主應用間的互通性，並簡化複雜代理系統的開發。
- **複雜流程編排：** 藉由組合多種 MCP 暴露的工具與資料來源，代理可編排高度複雜的多步驟流程。例如代理可從資料庫取得客戶資料、生成個人化行銷圖片、撰寫客製化郵件並寄送，整個流程由不同 MCP 服務協作完成。
- **IoT 裝置控制：** MCP 可促進 LLM 與物聯網（IoT）裝置互動。代理可用 MCP 發送命令給智慧家電、工業感測器或機器人，實現自然語言控制與自動化。
- **金融服務自動化：** 在金融領域，MCP 可讓 LLM 與各種金融資料來源、交易平台或合規系統互動。代理可分析市場資料、執行交易、提供個人化財務建議，或自動化法規報告，同時維持安全且標準化的通訊。

In short, the Model Context Protocol (MCP) enables agents to access real-time information from databases, APIs, and web resources. It also allows agents to perform actions like sending emails, updating records, controlling devices, and executing complex tasks by integrating and processing data from various sources. Additionally, MCP supports media generation tools for AI applications.

總之，模型上下文協定（MCP）讓代理可存取資料庫、API 與網路資源的即時資訊。它也讓代理能透過整合與處理多元來源的資料，執行寄信、更新紀錄、控制裝置與複雜任務等行動。此外，MCP 也支援 AI 應用中的媒體生成工具。

Hands-On Code Example with ADK

ADK 實作程式碼範例

This section outlines how to connect to a local MCP server that provides file system operations, enabling an ADK agent to interact with the local file system.

本節說明如何連接提供檔案系統操作的本地 MCP 伺服器，讓 ADK 代理能與本地檔案系統互動。

Agent Setup with MCPToolset

使用 MCPToolset 的代理設定

To configure an agent for file system interaction, an `agent.py` file must be created (e.g., at `./adk_agent_samples/mcp_agent/agent.py`). The `MCPToolset` is

instantiated within the `tools` list of the `LlmAgent` object. It is crucial to replace `"/path/to/your/folder"` in the `args` list with the absolute path to a directory on the local system that the MCP server can access. This directory will be the root for the file system operations performed by the agent.

要設定代理與檔案系統互動，必須建立 `agent.py` 檔案（例如 `./adk_agent_samples/mcp_agent/agent.py`）。`MCPToolset` 會在 `LlmAgent` 物件的 `tools` 列表中初始化。務必將 `args` 列表中的 `"/path/to/your/folder"` 替換為本地系統中 MCP 伺服器可存取的資料夾絕對路徑。此目錄將成為代理執行檔案系統操作的根目錄。

```
import os

from google.adk.agents import LlmAgent
from google.adk.tools.mcp_tool.mcp_toolset import MCPToolset,
StdioServerParameters


# Create a reliable absolute path to a folder named 'mcp_managed_files'
# within the same directory as this agent script.
# This ensures the agent works out-of-the-box for demonstration.
# For production, you would point this to a more persistent and secure
location.
TARGET_FOLDER_PATH = os.path.join(
    os.path.dirname(os.path.abspath(__file__)),
    "mcp_managed_files",
)

# Ensure the target directory exists before the agent needs it.
os.makedirs(TARGET_FOLDER_PATH, exist_ok=True)

root_agent = LlmAgent(
    model="gemini-2.0-flash",
    name="filesystem_assistant_agent",
    instruction=(
        "Help the user manage their files. You can list files, read files, and
write files. "
        f"You are operating in the following directory: {TARGET_FOLDER_PATH}"
    ),
    tools=[
        MCPToolset(
            connection_params=StdioServerParameters(
                command="npx",
                args=[
                    "-y",  # Argument for npx to auto-confirm install
                    "@modelcontextprotocol/server-filesystem",
```

```
                    # This MUST be an absolute path to a folder.
                    TARGET_FOLDER_PATH,
                ],
            ),
            # Optional: You can filter which tools from the MCP server are
exposed.
            # For example, to only allow reading:
            # tool_filter=['list_directory', 'read_file']
        )
    ],
)
```

`npx` (Node Package Execute), bundled with npm (Node Package Manager) versions 5.2.0 and later, is a utility that enables direct execution of Node.js packages from the npm registry. This eliminates the need for global installation. In essence, `npx` serves as an npm package runner, and it is commonly used to run many community MCP servers, which are distributed as Node.js packages.

`npx`（Node Package Execute）隨 npm（Node Package Manager）5.2.0 之後版本提供，是一種可直接執行 npm 註冊表中的 Node.js 套件的工具，免去全域安裝需求。本質上，`npx` 是 npm 套件執行器，常用於執行以 Node.js 套件發佈的社群 MCP 伺服器。

Creating an `__init__.py` file is necessary to ensure the agent.py file is recognized as part of a discoverable Python package for the Agent Development Kit (ADK). This file should reside in the same directory as agent.py.

需要建立 `__init__.py` 檔，讓 agent.py 被 ADK 視為可發現的 Python 套件的一部分。此檔案應放在與 agent.py 相同的目錄中。

```
# ./adk_agent_samples/mcp_agent/__init__.py
from . import agent
```

Certainly, other supported commands are available for use. For example, connecting to python3 can be achieved as follows:

當然，也有其他支援的指令可用。例如，可透過以下方式連接 python3：

```
connection_params = StdioConnectionParams(
    server_params={
        "command": "python3",
        "args": ["./agent/mcp_server.py"],
        "env": {
            "SERVICE_ACCOUNT_PATH": SERVICE_ACCOUNT_PATH,
            "DRIVE_FOLDER_ID": DRIVE_FOLDER_ID,
```

```
        },
    }
)
```

UVX, in the context of Python, refers to a command-line tool that utilizes uv to execute commands in a temporary, isolated Python environment. Essentially, it allows you to run Python tools and packages without needing to install them globally or within your project's environment. You can run it via the MCP server.

在 Python 情境中，UVX 指的是使用 uv 在臨時、隔離的 Python 環境中執行命令的命令列工具。它讓你無需全域或專案內安裝，就能執行 Python 工具與套件。你可以透過 MCP 伺服器執行它。

```
connection_params = StdioConnectionParams(
    server_params={
        "command": "uvx",
        "args": ["mcp-google-sheets@latest"],
        "env": {
            "SERVICE_ACCOUNT_PATH": SERVICE_ACCOUNT_PATH,
            "DRIVE_FOLDER_ID": DRIVE_FOLDER_ID,
        },
    }
)
```

Once the MCP Server is created, the next step is to connect to it.

當 MCP 伺服器建立後，下一步就是連線。

Connecting the MCP Server with ADK Web

透過 ADK Web 連接 MCP 伺服器

To begin, execute 'adk web'. Navigate to the parent directory of mcp_agent (e.g., adk_agent_samples) in your terminal and run:

首先執行 `adk web`。在終端機中切到 mcp_agent 的上層目錄（例如 adk_agent_samples），並執行：

```
cd ./adk_agent_samples # Or your equivalent parent directory
adk web
```

Once the ADK Web UI has loaded in your browser, select the `filesystem_assistant_agent` from the agent menu. Next, experiment with prompts such as:

當 ADK Web UI 在瀏覽器中載入後，從代理選單選擇 `filesystem_assistant_agent`。接著可用以下提示測試：

- “Show me the contents of this folder.”
- “Read the `sample.txt` file.” (This assumes `sample.txt` is located at `TARGET_FOLDER_PATH`.)
- “What’s in `another_file.md`?”
- 「顯示這個資料夾的內容。」
- 「讀取 `sample.txt` 檔案。」（假設 `sample.txt` 位於 `TARGET_FOLDER_PATH`。）
- 「`another_file.md` 裡有什麼？」

Creating an MCP Server with FastMCP

使用 FastMCP 建立 MCP 伺服器

FastMCP is a high-level Python framework designed to streamline the development of MCP servers. It provides an abstraction layer that simplifies protocol complexities, allowing developers to focus on core logic.

FastMCP 是一個高階 Python 框架，旨在簡化 MCP 伺服器開發。它提供抽象層以簡化協定複雜度，讓開發者專注於核心邏輯。

The library enables rapid definition of tools, resources, and prompts using simple Python decorators. A significant advantage is its automatic schema generation, which intelligently interprets Python function signatures, type hints, and documentation strings to construct necessary AI model interface specifications. This automation minimizes manual configuration and reduces human error.

此函式庫可透過簡單的 Python 裝飾器快速定義工具、資源與提示。其重大優勢是自動生成 schema，能智慧解讀 Python 函式簽名、型別提示與文件字串，建立必要的 AI 模型介面規格。這種自動化能減少人工設定並降低錯誤。

Beyond basic tool creation, FastMCP facilitates advanced architectural patterns like server composition and proxying. This enables modular development of complex, multi-component systems and seamless integration of existing services into an AI-accessible framework. Additionally, FastMCP includes optimizations for efficient, distributed, and scalable AI-driven applications.

除了基本工具建立外，FastMCP 也支援伺服器組合與代理等進階架構模式，使複雜多元件系統能以模組化方式開發，並將既有服務無縫整合到可由 AI 存取的框架中。此外，FastMCP 也包含對高效率、分散式與可擴展 AI 應用的最佳化。

Server setup with FastMCP

使用 FastMCP 的伺服器設定

To illustrate, consider a basic “greet” tool provided by the server. ADK agents and other MCP clients can interact with this tool using HTTP once it is active

以伺服器提供的基本「greet」工具為例。當它啟動後，ADK 代理與其他 MCP 用戶端可透過 HTTP 互動

```
# fastmcp_server.py
# This script demonstrates how to create a simple MCP server using FastMCP.
# It exposes a single tool that generates a greeting.
# 1. Make sure you have FastMCP installed:
# pip install fastmcp

from fastmcp import FastMCP, Client


# Initialize the FastMCP server.
mcp_server = FastMCP()


# Define a simple tool function.
# The `@mcp_server.tool` decorator registers this Python function as an MCP
tool.
# The docstring becomes the tool's description for the LLM.
@mcp_server.tool
def greet(name: str) -> str:
    """
    Generates a personalized greeting.

    Args:
        name: The name of the person to greet.

    Returns:
        A greeting string.
    """
    return f"Hello, {name}! Nice to meet you."


# Or if you want to run it from the script:
if __name__ == "__main__":
    mcp_server.run(
```

```
        transport="http",
        host="127.0.0.1",
        port=8000,
    )
```

This Python script defines a single function called greet, which takes a person's name and returns a personalized greeting. The `@tool()` decorator above this function automatically registers it as a tool that an AI or another program can use. The function's documentation string and type hints are used by FastMCP to tell the Agent how the tool works, what inputs it needs, and what it will return.

這段 Python 腳本定義了一個名為 greet 的函式，接收人名並回傳個人化問候。上方的 `@tool()` 裝飾器會自動將此函式註冊為 AI 或其他程式可用的工具。該函式的文件字串與型別提示會被 FastMCP 用來告訴代理此工具如何運作、需要哪些輸入與會回傳什麼。

When the script is executed, it starts the FastMCP server, which listens for requests on localhost:8000. This makes the greet function available as a network service. An agent could then be configured to connect to this server and use the greet tool to generate greetings as part of a larger task. The server runs continuously until it is manually stopped.

當腳本執行時，它會啟動 FastMCP 伺服器，在 localhost:8000 監聽請求，使 greet 函式成為網路服務。接著可將代理設定為連線到此伺服器，並在較大任務的一部分中使用 greet 工具產生問候。伺服器會持續運作直到手動停止。

Consuming the FastMCP Server with an ADK Agent

以 ADK 代理使用 FastMCP 伺服器

An ADK agent can be set up as an MCP client to use a running FastMCP server. This requires configuring HttpServerParameters with the FastMCP server's network address, which is usually http://localhost:8000.

ADK 代理可設定為 MCP 用戶端以使用運行中的 FastMCP 伺服器。這需要以 FastMCP 伺服器的網路位址設定 HttpServerParameters，通常為 http://localhost:8000。

A `tool_filter` parameter can be included to restrict the agent's tool usage to specific tools offered by the server, such as 'greet'. When prompted with a request like "Greet John Doe," the agent's embedded LLM identifies the 'greet' tool available via MCP, invokes it with the argument "John Doe," and returns

the server's response. This process demonstrates the integration of user-defined tools exposed through MCP with an ADK agent.

可加入 `tool_filter` 參數以限制代理只能使用伺服器提供的特定工具，如 'greet'。當收到像「Greet John Doe」的請求時，代理內嵌的 LLM 會識別 MCP 可用的 'greet' 工具，以 "John Doe" 作為參數呼叫，並回傳伺服器回應。此流程示範透過 MCP 暴露的自訂工具與 ADK 代理的整合。

To establish this configuration, an agent file (e.g., agent.py located in ./adk_agent_samples/fastmcp_client_agent/) is required. This file will instantiate an ADK agent and use HttpServerParameters to establish a connection with the operational FastMCP server.

為建立此設定，需要一個代理檔案（例如位於 ./adk_agent_samples/fastmcp_client_agent/ 的 agent.py）。該檔案會建立 ADK 代理，並使用 HttpServerParameters 與運作中的 FastMCP 伺服器連線。

```
# ./adk_agent_samples/fastmcp_client_agent/agent.py
import os

from google.adk.agents import LlmAgent
from google.adk.tools.mcp_tool.mcp_toolset import MCPToolset,
HttpServerParameters


# Define the FastMCP server's address.
# Make sure your fastmcp_server.py (defined previously) is running on this
port.
FASTMCP_SERVER_URL = "http://localhost:8000"

root_agent = LlmAgent(
    model="gemini-2.0-flash",  # Or your preferred model
    name="fastmcp_greeter_agent",
    instruction='You are a friendly assistant that can greet people by their
name. Use the "greet" tool.',
    tools=[
        MCPToolset(
            connection_params=HttpServerParameters(
                url=FASTMCP_SERVER_URL,
            ),
            # Optional: Filter which tools from the MCP server are exposed
            # For this example, we're expecting only 'greet'
            tool_filter=["greet"],
        )
```

```
    ],
)
```

The script defines an Agent named `fastmcp_greeter_agent` that uses a Gemini language model. It's given a specific instruction to act as a friendly assistant whose purpose is to greet people. Crucially, the code equips this agent with a tool to perform its task. It configures an MCPToolset to connect to a separate server running on localhost:8000, which is expected to be the FastMCP server from the previous example. The agent is specifically granted access to the greet tool hosted on that server. In essence, this code sets up the client side of the system, creating an intelligent agent that understands its goal is to greet people and knows exactly which external tool to use to accomplish it.

這段腳本定義名為 `fastmcp_greeter_agent` 的代理，使用 Gemini 語言模型，並給予友善助理的指令以向他人問候。關鍵在於程式碼為該代理配備了執行任務的工具：它設定 MCPToolset 連線到 localhost:8000 上的獨立伺服器（即前例的 FastMCP 伺服器），並明確授權使用該伺服器提供的 greet 工具。此程式碼本質上建立了系統的用戶端端，讓智慧代理知道目標是問候並知道要使用哪個外部工具來完成。

Creating an `__init__.py` file within the `fastmcp_client_agent` directory is necessary. This ensures the agent is recognized as a discoverable Python package for the ADK.

需要在 `fastmcp_client_agent` 目錄中建立 `__init__.py`，以確保該代理被 ADK 視為可發現的 Python 套件。

To begin, open a new terminal and run `python fastmcp_server.py` to start the FastMCP server. Next, go to the parent directory of `fastmcp_client_agent` (for example, `adk_agent_samples`) in your terminal and execute `adk web`. Once the ADK Web UI loads in your browser, select the `fastmcp_greeter_agent` from the agent menu. You can then test it by entering a prompt like "Greet John Doe." The agent will use the `greet` tool on your FastMCP server to create a response.

首先開啟新終端機並執行 `python fastmcp_server.py` 啟動 FastMCP 伺服器。接著在終端機中切到 `fastmcp_client_agent` 的上層目錄（例如 `adk_agent_samples`），執行 `adk web`。當 ADK Web UI 載入瀏覽器後，從代理選單選擇 `fastmcp_greeter_agent`。接著可輸入如「Greet John Doe」的提示測試。代理會使用 FastMCP 伺服器上的 `greet` 工具生成回覆。

At a Glance

一覽

What: To function as effective agents, LLMs must move beyond simple text generation. They require the ability to interact with the external environment to access current data and utilize external software. Without a standardized communication method, each integration between an LLM and an external tool or data source becomes a custom, complex, and non-reusable effort. This ad-hoc approach hinders scalability and makes building complex, interconnected AI systems difficult and inefficient.

什麼： 若要成為有效代理，LLM 必須超越單純文字生成，具備與外部環境互動、存取即時資料與使用外部軟體的能力。沒有標準化通訊方式時，每次 LLM 與外部工具或資料來源的整合都會變成客製、複雜且不可重用的工作。這種臨時做法妨礙擴展性，使複雜互聯 AI 系統的建置變得困難且低效。

Why: The Model Context Protocol (MCP) offers a standardized solution by acting as a universal interface between LLMs and external systems. It establishes an open, standardized protocol that defines how external capabilities are discovered and used. Operating on a client-server model, MCP allows servers to expose tools, data resources, and interactive prompts to any compliant client. LLM-powered applications act as these clients, dynamically discovering and interacting with available resources in a predictable manner. This standardized approach fosters an ecosystem of interoperable and reusable components, dramatically simplifying the development of complex agentic workflows.

為什麼： MCP 透過作為 LLM 與外部系統之間的通用介面提供標準化解法。它建立開放的標準協定，定義外部能力如何被發現與使用。MCP 以用戶端 / 伺服器模型運作，讓伺服器可向任何相容用戶端暴露工具、資料資源與互動提示。LLM 應用作為用戶端，能以可預期方式動態發現並互動可用資源。此標準化方法促成可互通、可重用的元件生態系，大幅簡化複雜代理流程的開發。

Rule of thumb: Use the Model Context Protocol (MCP) when building complex, scalable, or enterprise-grade agentic systems that need to interact with a diverse and evolving set of external tools, data sources, and APIs. It is ideal when interoperability between different LLMs and tools is a priority, and when agents require the ability to dynamically discover new capabilities without being

redeployed. For simpler applications with a fixed and limited number of predefined functions, direct tool function calling may be sufficient.

經驗法則： 當要建立需要與多樣且持續演變之外部工具、資料來源與 API 互動的複雜、可擴展或企業級代理系統時，應使用 MCP。若優先考量不同 LLM 與工具間的互通性，以及代理需能動態發現新能力而不重新部署，MCP 特別適合。對於功能固定且有限的簡單應用，直接工具函式呼叫可能已足夠。

Visual summary:

視覺摘要：

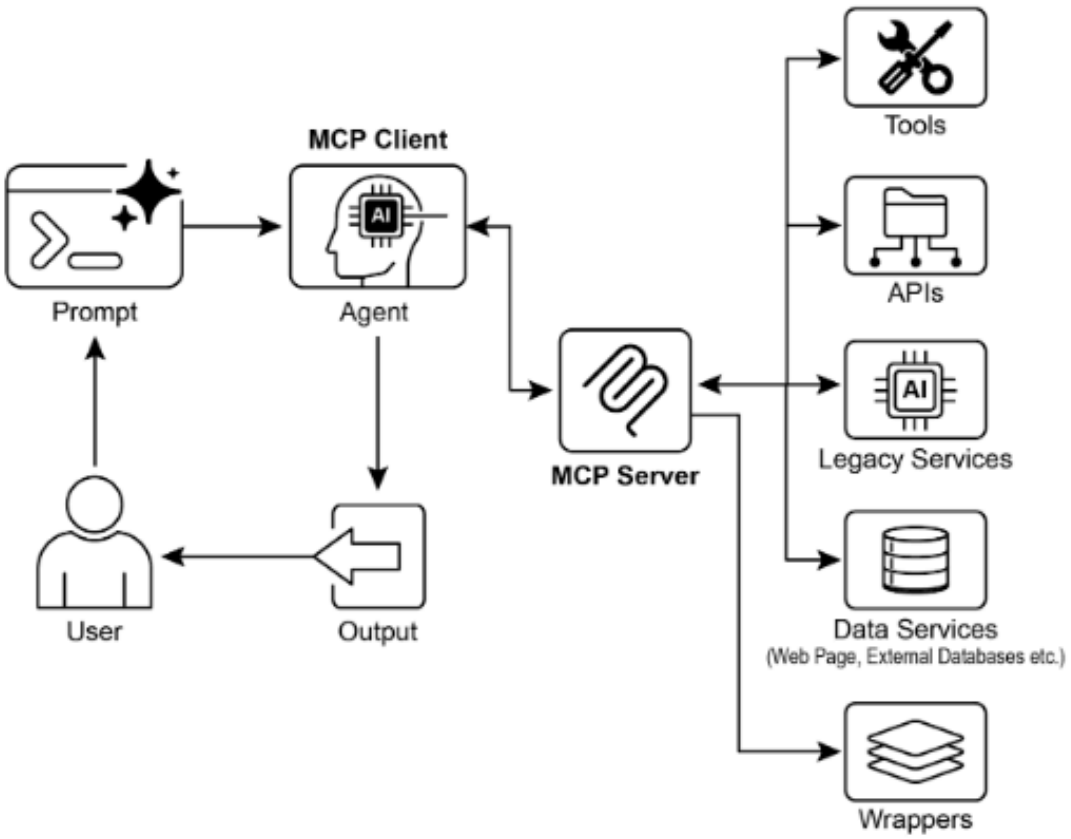

Figure 6: Model Context Protocol

Fig.1: Model Context protocol

圖 1：模型上下文協定

Key Takeaways

重點整理

These are the key takeaways:

以下為重點整理：

- The Model Context Protocol (MCP) is an open standard facilitating standardized communication between LLMs and external applications, data sources, and tools.
- It employs a client-server architecture, defining the methods for exposing and consuming resources, prompts, and tools.
- The Agent Development Kit (ADK) supports both utilizing existing MCP servers and exposing ADK tools via an MCP server.
- FastMCP simplifies the development and management of MCP servers, particularly for exposing tools implemented in Python.
- MCP Tools for Genmedia Services allows agents to integrate with Google Cloud's generative media capabilities (Imagen, Veo, Chirp 3 HD, Lyria).
- MCP enables LLMs and agents to interact with real-world systems, access dynamic information, and perform actions beyond text generation.
- MCP 是開放標準，促進 LLM 與外部應用、資料來源與工具之間的標準化通訊。
- 它採用用戶端 / 伺服器架構，定義資源、提示與工具的暴露與消費方式。
- ADK 支援使用既有 MCP 伺服器，也支援透過 MCP 伺服器暴露 ADK 工具。
- FastMCP 簡化 MCP 伺服器的開發與管理，特別適合暴露以 Python 實作的工具。
- MCP Tools for Genmedia Services 讓代理能整合 Google Cloud 的生成式媒體能力（Imagen、Veo、Chirp 3 HD、Lyria）。
- MCP 使 LLM 與代理能與真實系統互動、存取動態資訊，並執行超越文字生成的動作。

Conclusion

結論

The Model Context Protocol (MCP) is an open standard that facilitates communication between Large Language Models (LLMs) and external systems. It employs a client-server architecture, enabling LLMs to access resources, utilize prompts, and execute actions through standardized tools. MCP allows LLMs to interact with databases, manage generative media workflows, control IoT devices, and automate financial services. Practical examples demonstrate setting up agents to communicate with MCP servers, including filesystem

servers and servers built with FastMCP, illustrating its integration with the Agent Development Kit (ADK). MCP is a key component for developing interactive AI agents that extend beyond basic language capabilities.

模型上下文協定（MCP）是促進大型語言模型（LLM）與外部系統通訊的開放標準。它採用用戶端 / 伺服器架構，使 LLM 能透過標準化工具存取資源、使用提示並執行動作。MCP 讓 LLM 能與資料庫互動、管理生成式媒體流程、控制 IoT 裝置並自動化金融服務。實作範例示範如何設定代理與 MCP 伺服器通訊，包括檔案系統伺服器與 FastMCP 建置的伺服器，呈現其與 ADK 的整合方式。MCP 是開發超越基本語言能力的互動式 AI 代理的關鍵元件。

References

參考資料

1. Model Context Protocol (MCP) Documentation. (Latest). Model Context Protocol (MCP). https://google.github.io/adk-docs/mcp/
2. FastMCP Documentation. FastMCP. https://github.com/jlowin/fastmcp
3. MCP Tools for Genmedia Services. MCP Tools for Genmedia Services. https://google.github.io/adk-docs/mcp/#mcp-servers-for-google-cloud-genmedia
4. MCP Toolbox for Databases Documentation. (Latest). MCP Toolbox for Databases. https://google.github.io/adk-docs/mcp/databases/
5. Model Context Protocol（MCP）文件（最新版）。Model Context Protocol (MCP)。 https://google.github.io/adk-docs/mcp/
6. FastMCP 文件。FastMCP。 https://github.com/jlowin/fastmcp
7. MCP Tools for Genmedia Services。MCP Tools for Genmedia Services。 https://google.github.io/adk-docs/mcp/#mcp-servers-for-google-cloud-genmedia
8. MCP Toolbox for Databases 文件（最新版）。MCP Toolbox for Databases。 https://google.github.io/adk-docs/mcp/databases/

Chapter 11: Goal Setting and Monitoring

第 11 章：目標設定與監控

For AI agents to be truly effective and purposeful, they need more than just the ability to process information or use tools; they need a clear sense of direction

and a way to know if they're actually succeeding. This is where the Goal Setting and Monitoring pattern comes into play. It's about giving agents specific objectives to work towards and equipping them with the means to track their progress and determine if those objectives have been met.

要讓 AI 代理真正有效且有目的，它們不僅要能處理資訊或使用工具，還需要清楚的方向感與判斷是否成功的方法。這就是「目標設定與監控」模式的用武之地。它的核心是給代理明確的目標，並提供追蹤進度與判斷目標是否達成的手段。

Goal Setting and Monitoring Pattern Overview

目標設定與監控模式概覽

Think about planning a trip. You don't just spontaneously appear at your destination. You decide where you want to go (the goal state), figure out where you are starting from (the initial state), consider available options (transportation, routes, budget), and then map out a sequence of steps: book tickets, pack bags, travel to the airport/station, board the transport, arrive, find accommodation, etc. This step-by-step process, often considering dependencies and constraints, is fundamentally what we mean by planning in agentic systems.

想像規劃一次旅行。你不會突然就出現在目的地，而是先決定想去的地方（目標狀態），確認起點（初始狀態），考量可用選項（交通、路線、預算），再規劃一連串步驟：訂票、收拾行李、前往機場 / 車站、搭乘交通工具、抵達、找住宿等。這種逐步且考慮依賴與限制的流程，就是代理系統中所謂的規劃本質。

In the context of AI agents, planning typically involves an agent taking a high-level objective and autonomously, or semi-autonomously, generating a series of intermediate steps or sub-goals. These steps can then be executed sequentially or in a more complex flow, potentially involving other patterns like tool use, routing, or multi-agent collaboration. The planning mechanism might involve sophisticated search algorithms, logical reasoning, or increasingly, leveraging the capabilities of large language models (LLMs) to generate plausible and effective plans based on their training data and understanding of tasks.

在 AI 代理的情境中，規劃通常指代理以高階目標為起點，自主或半自主產生一系列中間步驟或子目標。這些步驟可以依序執行，也可採更複雜的流程，並可能結合工具使用、路由或多代理協作等模式。規劃機制可能涉及進階搜尋演算法與邏輯推理，或日益仰賴 LLM 以其訓練資料與任務理解能力產生可行且有效的計畫。

A good planning capability allows agents to tackle problems that aren't simple, single-step queries. It enables them to handle multi-faceted requests, adapt to changing circumstances by replanning, and orchestrate complex workflows. It's a foundational pattern that underpins many advanced agentic behaviors, turning a simple reactive system into one that can proactively work towards a defined objective.

良好的規劃能力讓代理能處理非單一步驟的問題，得以應對多面向需求、在情勢變化時重新規劃，並編排複雜流程。這是一個支撐多項進階代理行為的基礎模式，能將單純的反應式系統轉變為可主動朝既定目標行動的系統。

Practical Applications & Use Cases

實務應用與使用情境

The Goal Setting and Monitoring pattern is essential for building agents that can operate autonomously and reliably in complex, real-world scenarios. Here are some practical applications:

目標設定與監控模式對於在複雜真實場景中自主且可靠運作的代理至關重要。以下是一些實務應用：

- **Customer Support Automation:** An agent's goal might be to "resolve customer's billing inquiry." It monitors the conversation, checks database entries, and uses tools to adjust billing. Success is monitored by confirming the billing change and receiving positive customer feedback. If the issue isn't resolved, it escalates.
- **Personalized Learning Systems:** A learning agent might have the goal to "improve students' understanding of algebra." It monitors the student's progress on exercises, adapts teaching materials, and tracks performance metrics like accuracy and completion time, adjusting its approach if the student struggles.
- **Project Management Assistants:** An agent could be tasked with "ensuring project milestone X is completed by Y date." It monitors task statuses, team communications, and resource availability, flagging delays and suggesting corrective actions if the goal is at risk.
- **Automated Trading Bots:** A trading agent's goal might be to "maximize portfolio gains while staying within risk tolerance." It continuously monitors market data, its current portfolio value, and risk indicators, executing trades

when conditions align with its goals and adjusting strategy if risk thresholds are breached.

- **Robotics and Autonomous Vehicles:** An autonomous vehicle's primary goal is "safely transport passengers from A to B." It constantly monitors its environment (other vehicles, pedestrians, traffic signals), its own state (speed, fuel), and its progress along the planned route, adapting its driving behavior to achieve the goal safely and efficiently.
- **Content Moderation:** An agent's goal could be to "identify and remove harmful content from platform X." It monitors incoming content, applies classification models, and tracks metrics like false positives/negatives, adjusting its filtering criteria or escalating ambiguous cases to human reviewers.
- **客服自動化：** 代理的目標可能是「解決客戶帳單問題」。它監控對話、查詢資料庫並使用工具調整帳務。成功的判斷依賴帳務變更確認與正向回饋；若未解決則升級處理。
- **個人化學習系統：** 學習代理的目標可能是「提升學生對代數的理解」。它監控學生練習進度、調整教材，並追蹤正確率與完成時間等績效指標，若學生遇到困難則調整教學方式。
- **專案管理助理：** 代理可被指派「確保專案里程碑 X 於 Y 日期完成」。它監控任務狀態、團隊溝通與資源可用性，若目標有風險則標示延遲並建議修正行動。
- **自動化交易機器人：** 交易代理的目標可能是「在風險容忍範圍內最大化投資組合收益」。它持續監控市場資料、目前投資組合價值與風險指標，當條件符合目標時執行交易，並在風險閾值被突破時調整策略。
- **機器人與自駕車：** 自主車的主要目標是「安全地將乘客從 A 送到 B」。它持續監控環境（其他車輛、行人、號誌）、自身狀態（速度、油量）與沿路進度，並調整駕駛行為以安全且高效達成目標。
- **內容審核：** 代理的目標可能是「識別並移除平台 X 的有害內容」。它監控流入內容、套用分類模型並追蹤誤判率等指標，依此調整篩選準則或將模糊案例升級給人工審查。

This pattern is fundamental for agents that need to operate reliably, achieve specific outcomes, and adapt to dynamic conditions, providing the necessary framework for intelligent self-management.

此模式是需要可靠運作、達成特定成果並適應動態條件的代理之基礎，提供智能自我管理所需的框架。

Hands-On Code Example

實作程式碼範例

To illustrate the Goal Setting and Monitoring pattern, we have an example using LangChain and OpenAI APIs. This Python script outlines an autonomous AI agent engineered to generate and refine Python code. Its core function is to produce solutions for specified problems, ensuring adherence to user-defined quality benchmarks.

為了示範目標設定與監控模式，以下提供使用 LangChain 與 OpenAI API 的範例。這個 Python 腳本描述一個自主 AI 代理，負責產生並精煉 Python 程式碼。其核心功能是針對指定問題產出解法，並確保符合使用者定義的品質基準。

It employs a "goal-setting and monitoring" pattern where it doesn't just generate code once, but enters into an iterative cycle of creation, self-evaluation, and improvement. The agent's success is measured by its own AI-driven judgment on whether the generated code successfully meets the initial objectives. The ultimate output is a polished, commented, and ready-to-use Python file that represents the culmination of this refinement process.

它採用「目標設定與監控」模式，不是一次就生成程式碼，而是進入反覆的產生、由自我評估、再改進的循環。代理會用 AI 驅動的判斷來衡量產出是否符合初始目標。最終輸出是一個精煉、加註解且可直接使用的 Python 檔案，代表整個精修流程的成果。

Dependencies:

相依套件：

```
pip install langchain_openai openai python-dotenv .env file with key in OPENAI_API_KEY
```

You can best understand this script by imagining it as an autonomous AI programmer assigned to a project (see Fig. 1). The process begins when you hand the AI a detailed project brief, which is the specific coding problem it needs to solve.

可把此腳本想像成被指派專案的自主 AI 程式設計師（見圖 1）。流程始於你給 AI 一份詳細的專案說明，也就是它需要解決的具體程式問題。

```
# MIT License
# Copyright (c) 2025 Mahtab Syed
```

```
# https://www.linkedin.com/in/mahtabsyed/

"""
Hands-On Code Example - Iteration 2
-  To illustrate the Goal Setting and Monitoring pattern, we have an example
using LangChain and OpenAI APIs:

Objective: Build an AI Agent which can write code for a specified use case
based on specified goals:
-  Accepts a coding problem (use case) in code or can be as input.
-  Accepts a list of goals (e.g., "simple", "tested", "handles edge cases")
in code or can be input.
-  Uses an LLM (like GPT-4o) to generate and refine Python code until the
goals are met. (I am using max 5 iterations, this could be based on a set goal
as well)
-  To check if we have met our goals I am asking the LLM to judge this and
answer just True or False which makes it easier to stop the iterations.
-  Saves the final code in a .py file with a clean filename and a header
comment.
"""

import os
import random
import re
from pathlib import Path

from langchain_openai import ChatOpenAI
from dotenv import load_dotenv, find_dotenv


# 🔐 Load environment variables
_ = load_dotenv(find_dotenv())
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
if not OPENAI_API_KEY:
    raise EnvironmentError("❌ Please set the OPENAI_API_KEY environment
variable.")

# ✅ Initialize OpenAI model
print("📡 Initializing OpenAI LLM (gpt-4o)...")
llm = ChatOpenAI(
    model="gpt-4o",  # If you dont have access to got-4o use other OpenAI LLMs
    temperature=0.3,
    openai_api_key=OPENAI_API_KEY,
)
```

```python
# --- Utility Functions ---
def generate_prompt(
    use_case: str, goals: list[str], previous_code: str = "", feedback: str = ""
) -> str:
    print("📝 Constructing prompt for code generation...")
    base_prompt = f"""
You are an AI coding agent. Your job is to write Python code based on the following use case:

Use Case: {use_case}

Your goals are:
{chr(10).join(f"- {g.strip()}" for g in goals)}
"""
    if previous_code:
        print("🔄 Adding previous code to the prompt for refinement.")
        base_prompt += f"\nPreviously generated code:\n{previous_code}"
    if feedback:
        print("📋 Including feedback for revision.")
        base_prompt += f"\nFeedback on previous version:\n{feedback}\n"

    base_prompt += "\nPlease return only the revised Python code. Do not include comments or explanations outside the code."
    return base_prompt


def get_code_feedback(code: str, goals: list[str]) -> str:
    print("🔍 Evaluating code against the goals...")
    feedback_prompt = f"""
You are a Python code reviewer. A code snippet is shown below. Based on the following goals:

{chr(10).join(f"- {g.strip()}" for g in goals)}

Please critique this code and identify if the goals are met. Mention if improvements are needed for clarity, simplicity, correctness, edge case handling, or test coverage.

Code:
{code}
"""
    return llm.invoke(feedback_prompt)


def goals_met(feedback_text: str, goals: list[str]) -> bool:
```

```
    """
    Uses the LLM to evaluate whether the goals have been met based on the
feedback text.
    Returns True or False (parsed from LLM output).
    """
    review_prompt = f"""
You are an AI reviewer.

Here are the goals:
{chr(10).join(f"- {g.strip()}" for g in goals)}

Here is the feedback on the code:
\"\"\"
{feedback_text}
\"\"\"

Based on the feedback above, have the goals been met?

Respond with only one word: True or False.
"""
    response = llm.invoke(review_prompt).content.strip().lower()
    return response == "true"


def clean_code_block(code: str) -> str:
    lines = code.strip().splitlines()
    if lines and lines[0].strip().startswith("```"):
        lines = lines[1:]
    if lines and lines[-1].strip() == "```":
        lines = lines[:-1]
    return "\n".join(lines).strip()


def add_comment_header(code: str, use_case: str) -> str:
    comment = f"# This Python program implements the following use case:\n#
{use_case.strip()}\n"
    return comment + "\n" + code


def to_snake_case(text: str) -> str:
    text = re.sub(r"[^a-zA-Z0-9 ]", "", text)
    return re.sub(r"\s+", "_", text.strip().lower())


def save_code_to_file(code: str, use_case: str) -> str:
    print("💾 Saving final code to file...")
```

```
    summary_prompt = (
        f"Summarize the following use case into a single lowercase word or
phrase, "
        f"no more than 10 characters, suitable for a Python filename:
\n\n{use_case}"
    )
    raw_summary = llm.invoke(summary_prompt).content.strip()
    short_name = re.sub(r"[^a-zA-Z0-9_]", "", raw_summary.replace(" ",
"_").lower())[:10]

    random_suffix = str(random.randint(1000, 9999))
    filename = f"{short_name}_{random_suffix}.py"
    filepath = Path.cwd() / filename

    with open(filepath, "w") as f:
        f.write(code)

    print(f"✅ Code saved to: {filepath}")
    return str(filepath)


# --- Main Agent Function ---
def run_code_agent(use_case: str, goals_input: str, max_iterations: int = 5) -
> str:
    goals = [g.strip() for g in goals_input.split(",")]

    print(f"\n🎯 Use Case: {use_case}")
    print("🎯 Goals:")
    for g in goals:
        print(f"  - {g}")

    previous_code = ""
    feedback = ""

    for i in range(max_iterations):
        print(f"\n=== 🔁 Iteration {i + 1} of {max_iterations} ===")
        prompt = generate_prompt(
            use_case,
            goals,
            previous_code,
            feedback if isinstance(feedback, str) else feedback.content,
        )

        print("🚧 Generating code...")
        code_response = llm.invoke(prompt)
```

```
        raw_code = code_response.content.strip()
        code = clean_code_block(raw_code)
        print("\n📄 Generated Code:\n" + "-" * 50 + f"\n{code}\n" + "-" * 50)

        print("\n📤 Submitting code for feedback review...")
        feedback = get_code_feedback(code, goals)
        feedback_text = feedback.content.strip()
        print("\n📥 Feedback Received:\n" + "-" * 50 + f"\n{feedback_text}\n"
+ "-" * 50)

        if goals_met(feedback_text, goals):
            print("✅ LLM confirms goals are met. Stopping iteration.")
            break

        print("🛠️ Goals not fully met. Preparing for next iteration...")
        previous_code = code

    final_code = add_comment_header(code, use_case)
    return save_code_to_file(final_code, use_case)


# --- CLI Test Run ---
if __name__ == "__main__":
    print("\n🧠 Welcome to the AI Code Generation Agent")

    # Example 1
    use_case_input = "Write code to find BinaryGap of a given positive
integer"
    goals_input = "Code simple to understand, Functionally correct, Handles
comprehensive edge cases, Takes positive integer input only, prints the
results with few examples"
    run_code_agent(use_case_input, goals_input)

    # Example 2
    # use_case_input = "Write code to count the number of files in current
directory and all its nested sub directories, and print the total count"
    # goals_input = (
    #     "Code simple to understand, Functionally correct, Handles
comprehensive edge cases, Ignore recommendations for performance, Ignore
recommendations for test suite use like unittest or pytest"
    # )
    # run_code_agent(use_case_input, goals_input)

    # Example 3
    # use_case_input = "Write code which takes a command line input of a word
doc or docx file and opens it and counts the number of words, and characters
```

```
in it and prints all"
    # goals_input = "Code simple to understand, Functionally correct, Handles
edge cases"
    # run_code_agent(use_case_input, goals_input)
```

Along with this brief, you provide a strict quality checklist, which represents the objectives the final code must meet—criteria like “the solution must be simple,” “it must be functionally correct,” or “it needs to handle unexpected edge cases.”

在這份說明之外，你還會提供一份嚴格的品質檢核表，代表最終程式碼必須達成的目標，例如「解法必須簡單」、「必須功能正確」，或「需要處理意外的邊界情況」。

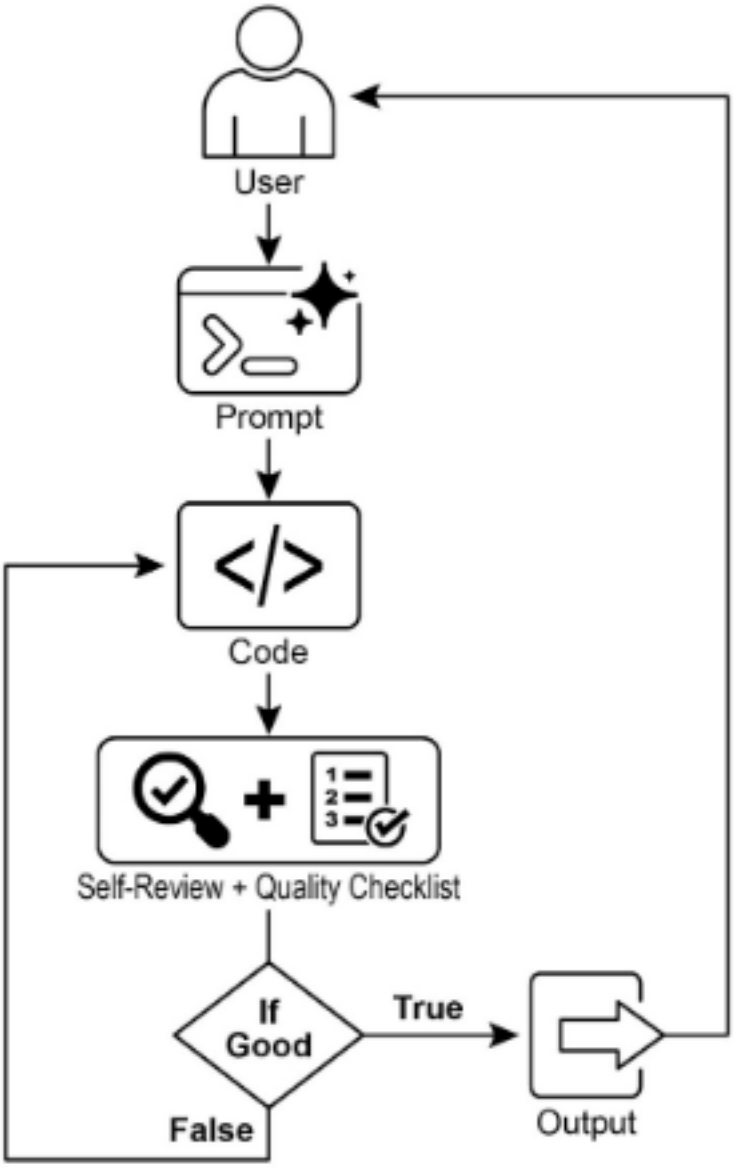

Figure 7: Goal Setting and Monitor example

Fig.1: Goal Setting and Monitor example

圖 1：目標設定與監控範例

With this assignment in hand, the AI programmer gets to work and produces its first draft of the code. However, instead of immediately submitting this initial version, it pauses to perform a crucial step: a rigorous self-review. It meticulously compares its own creation against every item on the quality checklist you provided, acting as its own quality assurance inspector. After this inspection, it renders a simple, unbiased verdict on its own progress: "True" if the work meets all standards, or "False" if it falls short.

拿到這項任務後，AI 程式設計師開始工作並產出第一版程式碼。但它不會立刻提交，而是先進行關鍵步驟：嚴格自我審查。它會仔細把自己的成果與品質檢核表逐一比對，充當自己的品質檢查員。檢查後，它會給出簡單且公正的判斷：若達標則為「True」，未達標則為「False」。

If the verdict is "False," the AI doesn't give up. It enters a thoughtful revision phase, using the insights from its self-critique to pinpoint the weaknesses and intelligently rewrite the code. This cycle of drafting, self-reviewing, and refining continues, with each iteration aiming to get closer to the goals. This process repeats until the AI finally achieves a "True" status by satisfying every requirement, or until it reaches a predefined limit of attempts, much like a developer working against a deadline. Once the code passes this final inspection, the script packages the polished solution, adding helpful comments and saving it to a clean, new Python file, ready for use.

若判定為「False」，AI 不會放棄，而會進入深思熟慮的修訂階段，利用自我評估的洞察找出弱點並智慧重寫程式碼。這種撰寫、審查與精修的循環持續進行，每次迭代都以更接近目標為目的。此流程會一直重複，直到 AI 最終以滿足所有要求取得「True」結果，或達到預設嘗試次數上限，就像面對期限的開發者。當程式碼通過最終檢查後，腳本會封裝精修後的解法，加上有用註解，並存成乾淨的新 Python 檔案，供直接使用。

Caveats and Considerations: It is important to note that this is an exemplary illustration and not production-ready code. For real-world applications, several factors must be taken into account. An LLM may not fully grasp the intended meaning of a goal and might incorrectly assess its performance as successful. Even if the goal is well understood, the model may hallucinate. When the same

LLM is responsible for both writing the code and judging its quality, it may have a harder time discovering it is going in the wrong direction.

注意事項與考量： 必須注意，這只是示範說明，並非可直接用於生產環境的程式碼。實際應用中需考慮多項因素。LLM 可能無法完全理解目標的意圖，並錯誤判斷其表現為成功。即使理解正確，模型仍可能產生幻覺。當同一個 LLM 同時負責寫程式與評估品質時，也可能更難察覺它正朝錯誤方向前進。

Ultimately, LLMs do not produce flawless code by magic; you still need to run and test the produced code. Furthermore, the “monitoring” in the simple example is basic and creates a potential risk of the process running forever.

最終，LLM 並不會神奇地產出完美程式碼；你仍必須執行並測試產出結果。此外，這個簡化範例中的「監控」相當基礎，可能導致流程無限迴圈的風險。

```
Act as an expert code reviewer with a deep commitment to producing clean, correct, and simple code. Your core mission is to eliminate code "hallucinations" by ensuring every suggestion is grounded in reality and best practices. When I provide you with a code snippet, I want you to: -- Identify and Correct Errors: Point out any logical flaws, bugs, or potential runtime errors. -- Simplify and Refactor: Suggest changes that make the code more readable, efficient, and maintainable without sacrificing correctness. -- Provide Clear Explanations: For every suggested change, explain why it is an improvement, referencing principles of clean code, performance, or security. -- Offer Corrected Code: Show the "before" and "after" of your suggested changes so the improvement is clear. Your feedback should be direct, constructive, and always aimed at improving the quality of the code.
```

A more robust approach involves separating these concerns by giving specific roles to a crew of agents. For instance, I have built a personal crew of AI agents using Gemini where each has a specific role:

更穩健的做法是將這些職責分離，為代理團隊指派明確角色。例如，我用 Gemini 建立了一個個人 AI 代理團隊，各自有明確分工：

- The Peer Programmer: Helps write and brainstorm code.
- The Code Reviewer: Catches errors and suggests improvements.
- The Documenter: Generates clear and concise documentation.
- The Test Writer: Creates comprehensive unit tests.
- The Prompt Refiner: Optimizes interactions with the AI.
- 同儕程式設計師：協助撰寫程式並腦力激盪。
- 程式碼審查者：抓出錯誤並提出改進建議。

- 文件撰寫者：產出清楚且簡潔的文件。
- 測試撰寫者：建立完整的單元測試。
- 提示精修者：最佳化與 AI 的互動。

In this multi-agent system, the Code Reviewer, acting as a separate entity from the programmer agent, has a prompt similar to the judge in the example, which significantly improves objective evaluation. This structure naturally leads to better practices, as the Test Writer agent can fulfill the need to write unit tests for the code produced by the Peer Programmer.

在此多代理系統中，程式碼審查者作為不同於程式設計代理的獨立角色，使用與範例裁判相似的提示，大幅提升客觀評估。此結構自然導向更佳實作，因為測試撰寫者代理可為同儕程式設計師產出的程式碼撰寫單元測試。

I leave to the interested reader the task of adding these more sophisticated controls and making the code closer to production-ready.

我將更精細的控制與更接近生產級的改造留給有興趣的讀者。

At a Glance

一覽

What: AI agents often lack a clear direction, preventing them from acting with purpose beyond simple, reactive tasks. Without defined objectives, they cannot independently tackle complex, multi-step problems or orchestrate sophisticated workflows. Furthermore, there is no inherent mechanism for them to determine if their actions are leading to a successful outcome. This limits their autonomy and prevents them from being truly effective in dynamic, real-world scenarios where mere task execution is insufficient.

什麼： AI 代理常缺乏明確方向，無法超越簡單的反應式任務而有目的地行動。沒有明確目標時，代理無法獨立處理複雜多步驟問題或編排精密流程。此外，代理缺乏內建機制判斷其行動是否導向成功，這限制了其自主性，也使其在動態真實場景中無法真正有效，僅執行任務並不足夠。

Why: The Goal Setting and Monitoring pattern provides a standardized solution by embedding a sense of purpose and self-assessment into agentic systems. It involves explicitly defining clear, measurable objectives for the agent to achieve. Concurrently, it establishes a monitoring mechanism that continuously tracks the agent's progress and the state of its environment against these goals. This

creates a crucial feedback loop, enabling the agent to assess its performance, correct its course, and adapt its plan if it deviates from the path to success. By implementing this pattern, developers can transform simple reactive agents into proactive, goal-oriented systems capable of autonomous and reliable operation.

為什麼： 目標設定與監控模式透過在代理系統中嵌入目的感與自我評估，提供標準化解法。它明確定義代理要達成的清晰且可衡量目標，同時建立監控機制，持續追蹤代理進度與環境狀態是否符合目標。這形成關鍵回饋迴圈，使代理能評估表現、修正路徑，並在偏離成功路徑時調整計畫。導入此模式可把簡單的反應式代理轉為主動、目標導向且可自主可靠運作的系統。

Rule of thumb: Use this pattern when an AI agent must autonomously execute a multi-step task, adapt to dynamic conditions, and reliably achieve a specific, high-level objective without constant human intervention.

經驗法則： 當 AI 代理需在無持續人為介入下，自主執行多步驟任務、適應動態條件並可靠達成特定高階目標時，應使用此模式。

Visual summary:

視覺摘要：

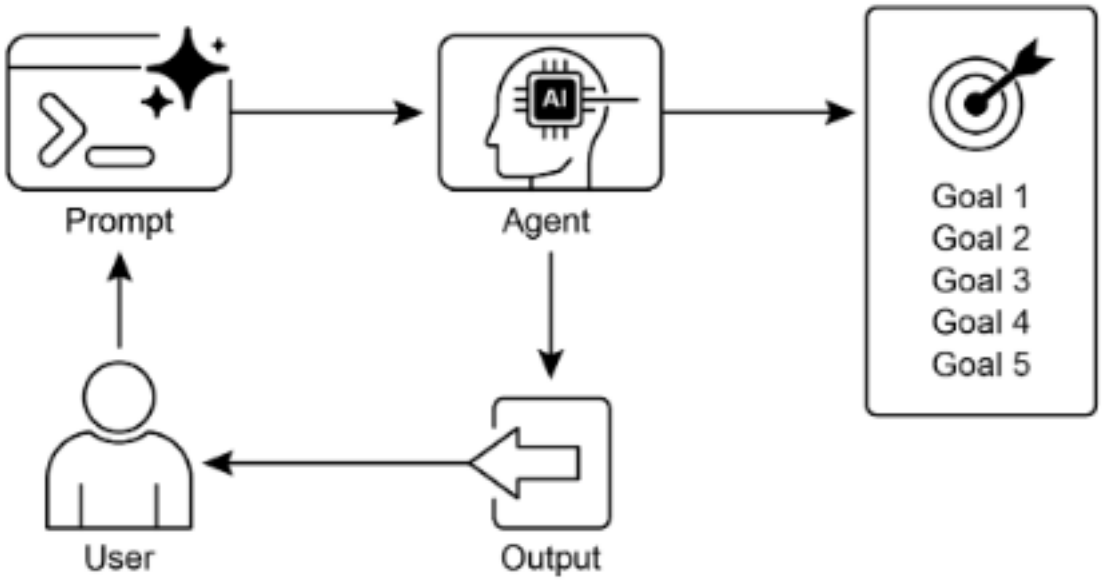

Figure 8: Goal Design Pattern

Fig.2: Goal design patterns

圖 2：目標設計模式

Key takeaways

重點整理

Key takeaways include:

重點如下：

- Goal Setting and Monitoring equips agents with purpose and mechanisms to track progress.
- Goals should be specific, measurable, achievable, relevant, and time-bound (SMART).
- Clearly defining metrics and success criteria is essential for effective monitoring.
- Monitoring involves observing agent actions, environmental states, and tool outputs.
- Feedback loops from monitoring allow agents to adapt, revise plans, or escalate issues.
- In Google's ADK, goals are often conveyed through agent instructions, with monitoring accomplished through state management and tool interactions.
- 目標設定與監控讓代理具備目的與追蹤進度的機制。
- 目標應具體、可衡量、可達成、相關且具時限（SMART）。
- 清楚定義指標與成功標準對有效監控至關重要。
- 監控包含觀察代理行動、環境狀態與工具輸出。
- 監控所形成的回饋迴圈可讓代理適應、修訂計畫或升級問題。
- 在 Google ADK 中，目標通常透過代理指令傳達，監控則透過 state 管理與工具互動完成。

Conclusion

結論

This chapter focused on the crucial paradigm of Goal Setting and Monitoring. I highlighted how this concept transforms AI agents from merely reactive systems into proactive, goal-driven entities. The text emphasized the importance of defining clear, measurable objectives and establishing rigorous monitoring procedures to track progress. Practical applications demonstrated how this

paradigm supports reliable autonomous operation across various domains, including customer service and robotics. A conceptual coding example illustrates the implementation of these principles within a structured framework, using agent directives and state management to guide and evaluate an agent's achievement of its specified goals. Ultimately, equipping agents with the ability to formulate and oversee goals is a fundamental step toward building truly intelligent and accountable AI systems.

本章聚焦於目標設定與監控這項關鍵範式，強調它如何將 AI 代理從純反應式系統轉變為主動、目標驅動的實體。文章強調明確可衡量目標與嚴謹監控程序的重要性，以追蹤進度。實務應用顯示，這一範式能在客戶服務、機器人等多個領域支援可靠的自主運作。概念性程式範例展示如何在結構化框架中實作這些原則，透過代理指令與 state 管理引導與評估代理達成指定目標。最終，賦予代理制定並監督目標的能力，是打造真正智慧且具責任 AI 系統的基礎一步。

References

參考資料

1. SMART Goals Framework. https://en.wikipedia.org/wiki/SMART_criteria
2. SMART 目標框架。 https://en.wikipedia.org/wiki/SMART_criteria