🚀 AI 實作課程完整教學（2025 夏季班）

1. 🧠 AI 基礎與應用探索

AI Agent 介紹

AI Agent（人工智慧代理人）是指能夠自主感知環境、做出決策並執行任務的人工智慧系統。AI Agent 通常具備感知（Perception）、推理（Reasoning）、學習（Learning）與行動（Action）等能力。這些代理人可以根據外部輸入（如語音、文字、影像等）進行分析，並根據目標自動調整行為。

- **感知（Perception）**：指 AI Agent 能夠接收並理解來自外部世界的資訊，例如語音辨識、影像辨識等。
- **推理（Reasoning）**：AI Agent 會根據已知資訊進行邏輯推斷，做出決策。
- **學習（Learning）**：AI Agent 能夠從經驗中學習，優化自身行為。
- **行動（Action）**：AI Agent 根據決策執行具體任務，例如回覆訊息、控制裝置等。

LLM 介紹

LLM（Large Language Model，大型語言模型）是一種基於深度學習的人工智慧模型，能夠理解與產生自然語言。LLM 透過大量語料訓練，學會語言結構、語意關聯，並能進行對話、寫作、翻譯等任務。

- **深度學習（Deep Learning）**：一種模仿人腦神經網路的機器學習方法，能自動從大量資料中學習特徵。
- **語料（Corpus）**：指用於訓練語言模型的大量文本資料。
- **語意（Semantics）**：語言中詞語、句子的意義。

AI Agent 與產業應用案例

AI Agent 已廣泛應用於客服、醫療、金融、教育等領域。例如：

- **客服機器人**：自動回應顧客問題，提升服務效率。
- **醫療助理**：協助醫生分析病例、提醒用藥。
- **智慧助理**：如行事曆提醒、語音助理等。

2. 🛠️ AI 專案規劃與初步實作

專案目標與需求分析

專案目標是指專案最終希望達成的成果。**需求分析**則是釐清使用者需求、功能規格與限制條件。

- **需求分析（Requirement Analysis）**：系統性地蒐集、分析並定義專案需求。
- **功能規格（Functional Specification）**：明確描述系統應具備的功能。

實作方法

1. **使用者訪談與需求蒐集**：
    - 透過開放式問題與主動聆聽，深入了解使用者痛點與期望。
    - 可建立使用者畫像（Persona）來具體化目標用戶輪廓。
2. **撰寫需求文件**：

    - 將蒐集到的需求整理成清晰、簡潔、可測試的條目。
    - 列出明確的功能清單與非功能需求（如效能、安全性）。
    - 可採用 MoSCoW 方法（Must have, Should have, Could have, Won't have）進行需求優先級排序。
3. **視覺化呈現**：
    - 使用表格整理需求細項。
    - 運用 UML 用例圖（Use Case Diagram）描述使用者與系統的互動情境。
        - **UML 用例圖範例（文字描述）**：
            - **用例名稱**：使用者查詢天氣資訊。
            - **參與者（Actor）**：使用者。
            - **系統**：AI 助理。
            - **主要流程**：
                1. 使用者向 AI 助理詢問：「倫敦今天天氣如何？」
                2. AI 助理識別使用者意圖（查詢天氣）及關鍵實體（地點：倫敦）。
                3. AI 助理呼叫外部天氣 API，傳入「倫敦」作為參數。
                4. 天氣 API 回傳倫敦的天氣資訊。
                5. AI 助理將天氣資訊整理後，以自然語言回覆給使用者。

功能模組拆解與關聯設計

將整體系統拆分為多個**模組（Module）**，每個模組負責特定功能。模組間的**關聯設計**則確保資料流與功能協作順暢。

- **模組（Module）**：系統中具備獨立功能的單元。
- **資料流（Data Flow）**：資料在系統中流動的路徑。

實作方法

1. **模組劃分**：
    - 根據功能職責，將系統拆分為數個高內聚（High Cohesion - 模組內部功能關聯緊密）且低耦合（Low Coupling - 模組之間依賴性低）的模組。
    - 可使用 UML 類別圖（Class Diagram）或元件圖（Component Diagram）來視覺化模組結構。
2. **定義模組職責與介面**：
    - 明確每個模組的輸入（Input）、處理邏輯（Processing）、輸出（Output）。
    - 設計模組間的資料交換介面（API），例如：
        - 若為 Web 服務，可遵循 RESTful API 設計原則。
        - 定義清晰的請求（Request）與回應（Response）資料格式（如 JSON）。
        - 考慮 API 版本管理。
3. **繪製資料流程圖**：
    - 使用 UML 活動圖（Activity Diagram）或序列圖（Sequence Diagram）來描述資料如何在不同模組間流動及處理順序。

基本對答功能實作

AI Agent 的基本對答功能，通常包含**自然語言理解（NLU, Natural Language Understanding）與自然語言生成（NLG, Natural Language Generation）**。

- **自然語言理解（NLU）**：AI 理解使用者輸入的語意。
- **自然語言生成（NLG）**：AI 產生自然且合適的回應。

實作方法與範例程式碼

以 Python 串接 OpenAI API (gpt-3.5-turbo 或更新模型) 為例。請注意，您需要先安裝 `openai` 套件 (`pip install openai`) 並設定您的 API 金鑰。

```
import openai
import os

# 最佳實踐是從環境變數讀取 API 金鑰
# client = openai.OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))
# 為了示範，這裡使用佔位符。請替換成您實際的金鑰管理方式。
# 強烈建議不要將 API 金鑰直接寫在程式碼中。
try:
    client = openai.OpenAI(api_key='YOUR_API_KEY_HERE') # 請替換成您的金鑰或使用 os.environ
except openai.OpenAIError as e:
    print(f"OpenAI API 金鑰設定錯誤: {e}")
    client = None

# 基本對答函式，並支援對話歷史記錄
def chat_with_gpt(prompt_text, conversation_history=None):
    if not client:
        return "OpenAI API 未正確設定，無法進行對話。", []

    if conversation_history is None:
        conversation_history = []

    # 將目前使用者的輸入加入到對話歷史中
    current_messages = conversation_history + [{"role": "user", "content": prompt_text}]

    try:
        response = client.chat.completions.create(
            model="gpt-3.5-turbo", # 或其他您想使用的模型
            messages=current_messages
        )
        assistant_reply = response.choices[0].message.content

        # 更新對話歷史，加入 AI 的回覆，以便下一輪對話使用
        updated_history = current_messages + [{"role": "assistant", "content": assistant_reply}]
        return assistant_reply, updated_history
    except openai.APIError as e:
        print(f"OpenAI API 呼叫錯誤: {e}")
        return f"抱歉，與 AI 對話時發生錯誤: {e}", current_messages # 回傳錯誤前的歷史記錄
    except Exception as e:
        print(f"發生預期外的錯誤: {e}")
        return "抱歉，發生未知錯誤。", current_messages
```

```
# 範例使用
# conversation_history_main = [] # 初始化對話歷史

# user_input1 = "請問什麼是 AI Agent?"
# answer1, conversation_history_main = chat_with_gpt(user_input1,
conversation_history_main)
# print("AI:", answer1)

# if "錯誤" not in answer1: # 只有在第一次成功後才繼續
#     user_input2 = "它有哪些主要能力？" # 追問
#     answer2, conversation_history_main = chat_with_gpt(user_input2,
conversation_history_main)
#     print("AI:", answer2)

# 注意：在實際應用中，請確保您的 API 金鑰安全存放，
# 不要硬編碼在程式中。建議使用環境變數或密鑰管理服務。
```

---

3. 🧾 提示詞設計與流程思維

設計有效 Prompt

Prompt（提示詞）是指引導 LLM 產生特定回應的輸入文字。設計良好的 Prompt 能提升 AI 回答的準確性與相關性。

- **Prompt Engineering（提示詞工程）**：設計、優化 Prompt 的技術。

實作方法與範例

1. **明確任務、角色與目標受眾**：
   - Prompt 範例：「你是一位資深天文學家，請向一位對天文學完全不了解的 10 歲孩童解釋什麼是黑洞，請使用簡單的比喻，並避免專業術語。」
2. **指定回答格式**：
   - Prompt 範例：「請用 JSON 格式列出三種常見的程式語言，包含名稱 (name)、主要用途 (primary_use) 和一個簡短的優點 (pro)。」
3. **提供範例 (Few-shot Prompting)**：給予 LLM 一或多個輸入輸出的範例，引導其學習期望的模式。
   - Prompt 範例：

```
將以下句子從隨意口吻轉為專業書面語：
口語：老闆，那個報告我明天再給你啦。
書面語：經理，關於您提及的報告，我預計將於明日提交。

口語：這東西超讚的，一定要試試！
書面語：此產品具有卓越的性能，強烈建議您體驗。

口語：今天天氣有夠爛。
書面語：
```

(期望 LLM 接著產生：「今日天候狀況不佳。」)

4. **引導思考過程 (Chain-of-Thought Prompting, CoT)**：要求 LLM 在回答前先「逐步思考」或「解釋其推理過程」，有助於處理複雜問題。
    - Prompt 範例：「問題：小明有 5 個蘋果，他給了小華 2 個，然後他又從市場買了 3 袋蘋果，每袋有 4 個。請問小明現在有幾個蘋果？請逐步思考並給出最終答案。」
        - (期望 LLM 回答類似：1. 小明原有 5 個蘋果。2. 給小華 2 個後，剩下 5 - 2 = 3 個。3. 他買了 3 袋，每袋 4 個，所以總共買了 3 * 4 = 12 個。4. 因此，小明現在有 3 + 12 = 15 個蘋果。最終答案是 15。)
5. **避免模糊與歧義**：
    - **不佳的 Prompt**：「告訴我關於狗的事情。」 (過於空泛)
    - **較佳的 Prompt**：「請說明拉布拉多犬作為家庭寵物的典型壽命、常見健康問題以及每日建議運動量。」 (具體、提供上下文)

UML 流程圖規劃

UML（Unified Modeling Language，統一建模語言）是一種用於描述系統結構與行為的標準化圖形語言。**流程圖（Flowchart）**則用於描述任務或資料的處理流程。

- **UML**：用於軟體設計的標準圖形語言。
- **流程圖（Flowchart）**：用圖形表示流程步驟的工具。

實作方法

1. 使用 draw.io、Lucidchart 等工具繪製流程圖。
2. 以「使用者發問→AI 理解→AI 回答」為例，畫出流程。

多階段問題處理與 LLM 串接策略

多階段問題處理指將複雜問題拆解為多個步驟，逐步引導 LLM 回答。**串接（Integration）**是指將 LLM 與其他系統或模組連結。

- **串接（Integration）**：將不同系統或模組連結協作。

實作方法與範例程式碼

延續前述 `chat_with_gpt` 函式（已包含對話歷史管理），以 Python 實現多階段對話：

```
# (假設 chat_with_gpt 函式已如上文定義，並能處理 conversation_history)
def multi_stage_conversation_improved():
    conversation_log = [] # 初始化空的對話歷史

    prompt1 = "請問您想了解哪一方面的 AI 技術？ (例如：機器學習、自然語言處理、電腦視覺)"
    ai_response1, conversation_log = chat_with_gpt(prompt1, conversation_log)
    print("AI:", ai_response1)

    if "錯誤" in ai_response1: return # 如果第一步出錯則終止
```

```
    user_reply1 = input("你: ")
    # user_reply1 已透過 chat_with_gpt 的下一次呼叫自動加入到 conversation_log
中

    prompt2 = f"關於您提到的 '{user_reply1}'，請提供三個主要的應用案例，並為每個案
例做一個簡短的說明。"
    ai_response2, conversation_log = chat_with_gpt(prompt2,
conversation_log)
    print("AI:", ai_response2)

    # 可以繼續進行更多階段的對話...
    # if "錯誤" not in ai_response2:
    #     prompt3 = "針對這些應用案例，目前有哪些主要的挑戰？"
    #     ai_response3, conversation_log = chat_with_gpt(prompt3,
conversation_log)
    #     print("AI:", ai_response3)

# 執行範例
# multi_stage_conversation_improved()
```

為何需要多階段處理？

- **降低複雜度**：將一個複雜的大問題拆解成一系列小問題，引導 LLM 逐步達成目標。
- **維持上下文**：在多輪對話中，LLM 需要理解之前的交流內容才能給出連貫且相關的回應。
- **引導使用者**：透過提問逐步釐清使用者模糊的需求。
- **整合外部工具**：在不同階段可能需要呼叫不同的 API 或工具 (Tool Using / Function Calling)。

---

4. 🧩 問題分類與回饋機制

問題分類模組

問題分類（Question Classification）是指將使用者問題自動歸類到不同主題或類型，便於後續處理。

- **分類器（Classifier）**：用於自動判斷資料所屬類別的模型。

實作方法與範例程式碼

方法一：傳統機器學習分類器 (以 scikit-learn 為例) 您需要先安裝 `scikit-learn` 套件 (`pip install scikit-learn`)。

```
from sklearn.feature_extraction.text import TfidfVectorizer # Tfidf 通常比
CountVectorizer 效果好一些
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split

# 範例訓練資料 (實際應用中需要更多、更高品質的資料)
data = [
    ("請問如何下訂單？", "訂單問題"),
```

```
    ("我的包裹什麼時候會到？", "訂單問題"),
    ("什麼是大型語言模型？", "AI知識"),
    ("AI Agent 有哪些實際應用？", "AI知識"),
    ("產品價格是多少？", "產品資訊"),
    ("這個商品有保固嗎？", "產品資訊"),
    ("你好", "閒聊"),
    ("今天天氣真好", "閒聊")
]
texts, labels = zip(*data)

# 切分訓練集與測試集
X_train, X_test, y_train, y_test = train_test_split(texts, labels,
test_size=0.25, random_state=42)

# 建立模型管線：TF-IDF 向量化 -> 多項式樸素貝氏分類器
model = make_pipeline(TfidfVectorizer(), MultinomialNB())

# 訓練模型
model.fit(X_train, y_train)

# 預測新問題
# new_questions = ["AI 可以用來做什麼?", "我想取消訂單", "這件衣服有其他顏色嗎？"]
# predictions = model.predict(new_questions)
# for q, p in zip(new_questions, predictions):
#     print(f"問題: '{q}' -> 分類結果: {p}")

# 評估模型 (可選)
# accuracy = model.score(X_test, y_test)
# print(f"模型準確率: {accuracy:.2f}")
```

注意：傳統機器學習分類器需要足夠的標註資料進行訓練，且對於訓練資料中未出現過的詞彙或問法可能表現不佳。

方法二：使用大型語言模型 (LLM) 進行零樣本 (Zero-shot) 或少樣本 (Few-shot) 分類 這種方法利用 LLM 本身的理解能力，通常不需要額外訓練資料，或只需要少量範例。

```
# (假設 chat_with_gpt 函式已如上文定義)
def classify_question_with_llm(question, categories, examples=None):
    """
    使用 LLM 對問題進行分類。
    :param question: 要分類的使用者問題。
    :param categories: 一個包含所有可能類別的列表。
    :param examples: (可選) 一個包含少樣本範例的列表，每個範例是 (問題, 類別) 的元
組。
    :return: 預測的類別名稱 (字串)。
    """
    prompt = "請將以下使用者問題分類到最適當的類別中。
"
    prompt += f"可選類別有：{', '.join(categories)}。

"
```

```
    if examples:
        prompt += "以下是一些範例：
"
        for ex_q, ex_cat in examples:
            prompt += f"問題：'{ex_q}' -> 類別：{ex_cat}
"
        prompt += "
"

    prompt += f"現在，請分類這個問題：'{question}'
"
    prompt += "類別："

    # 為了確保 LLM 只回傳類別名稱，可以調整 chat_with_gpt 或後續處理
    # 這裡假設 chat_with_gpt 能較好地遵循指示
    predicted_category, _ = chat_with_gpt(prompt)

    # 清理 LLM 可能回傳的多餘文字，只取最可能的類別
    # 這部分可能需要更複雜的後處理邏輯來確保準確性
    for cat in categories:
        if cat.lower() in predicted_category.lower():
            return cat
    return predicted_category.strip().split('
')[0] # 嘗試取第一行作為類別

# 範例使用
# user_q = "我的訂單好像寄丟了，可以幫我查一下嗎？"
# defined_cats = ["訂單問題", "產品資訊", "AI技術諮詢", "帳戶問題", "閒聊"]

# 少樣本範例 (可選)
# few_shot_examples = [
#     ("你們的退貨政策是什麼？", "產品資訊"),
#     ("忘記密碼了怎麼辦？", "帳戶問題")
# ]

# llm_category = classify_question_with_llm(user_q, defined_cats,
examples=few_shot_examples)
# print(f"使用者問題: '{user_q}'")
# print(f"LLM 分類結果: {llm_category}")
```

使用者回饋處理方式

回饋機制（Feedback Mechanism）是指收集並處理使用者意見，協助系統持續優化。

- **回饋（Feedback）**：使用者對系統的意見或建議。
- **優化（Optimization）**：持續改進系統效能或體驗。

實作方法

1. **設計回饋收集機制**：

    - **顯性回饋**：在 AI 回應後提供「讚/倒讚」按鈕、星級評分、或簡短的意見回饋表單（例如：「這個回答有幫助嗎？」、「您期望得到什麼樣的回答？」）。
    - **隱性回饋**：分析使用者行為，例如：使用者是否在得到 AI 回答後追問了相關問題（可能表示回答不夠完整）、是否很快結束對話（可能滿意，也可能不滿意）、是否修正了 AI 的回答等。
2. **儲存與結構化回饋資料**：
    - 將回饋與對應的對話內容（問題、AI 回答、時間戳、使用者 ID 等）關聯起來儲存。
    - 使用資料庫或日誌系統來管理回饋資料。
3. **定期分析與歸納回饋**：
    - 找出常見的錯誤類型、使用者不滿意的點、或 AI 理解困難的場景。
    - 將回饋分類，例如：回答不準確、語氣不當、資訊過時、未能理解問題等。
4. **迭代改進 AI 系統**：
    - **優化提示詞 (Prompt Engineering)**：根據回饋調整 Prompt，使其更清晰、更具引導性。
    - **調整模型參數**：如果使用的是可調參數的模型。
    - **更新知識庫**：若 AI 依賴外部知識庫，需根據回饋更新或補充知識。
    - **模型再訓練/微調 (Fine-tuning)**：對於傳統機器學習模型，可以使用收集到的負面案例進行再訓練；對於 LLM，若有足夠高品質的回饋資料，可以考慮進行微調 (但成本較高)。
    - **改善流程邏輯**：調整 AI Agent 的決策流程或與其他模組的互動方式。
5. **建立閉環回饋系統**：告知使用者其回饋已被收到或已用於改進，增加使用者參與感。

---

5. 🧠 記憶模組實作

記憶模組方法與測試

記憶模組（Memory Module）讓 AI Agent 能記住過往對話或事件，提升互動連貫性。常見方法有短期記憶（Short-term Memory）與長期記憶（Long-term Memory）。

- **短期記憶**：暫時保存近期資訊。
- **長期記憶**：持久保存重要資訊。

實作方法與範例程式碼

短期記憶 (Short-term Memory / Context Window)

LLM 本身就有所謂的「上下文視窗 (Context Window)」，這是其內建的短期記憶形式，能夠記住最近的對話內容。我們在 `chat_with_gpt` 函式中傳遞 `conversation_history` 就是在利用這個機制。

以下是一個更明確的短期記憶類別範例，可以整合到 Agent 中：

```
class ConversationBufferMemory:
    def __init__(self, max_tokens=2000, max_messages=10): # 限制記憶大小
        self.history = []
        self.max_tokens = max_tokens # 粗略的 token 限制 (實際 token 計算較複雜)
        self.max_messages = max_messages

    def add_message(self, role, content):
        self.history.append({"role": role, "content": content})
        self._prune_memory()
```

```
    def get_history(self):
        return self.history

    def _prune_memory(self):
        # 簡單的修剪策略：如果訊息過多，則移除最早的訊息
        while len(self.history) > self.max_messages:
            self.history.pop(0)
        # 更進階的策略可以考慮 token 數量，或保留系統訊息等

    def clear(self):
        self.history = []

# 範例使用
# short_term_mem = ConversationBufferMemory(max_messages=5)
# short_term_mem.add_message("user", "你好")
# short_term_mem.add_message("assistant", "你好，有什麼可以協助您的嗎？")
# short_term_mem.add_message("user", "我想查詢天氣")
# short_term_mem.add_message("assistant", "請問您想查詢哪個城市的天氣？")
# short_term_mem.add_message("user", "台北")
# short_term_mem.add_message("assistant", "正在為您查詢台北的天氣...") # 第六則訊息，會擠掉第一則
# short_term_mem.add_message("user", "謝謝") # 第七則，會擠掉第二則

# print(short_term_mem.get_history())
```

長期記憶 (Long-term Memory) 與 檢索增強生成 (Retrieval Augmented Generation, RAG)

長期記憶允許 AI Agent 存取並利用超出當前對話上下文的廣泛知識。RAG 是一種常見的實現方式。

RAG 概念：

1. **知識庫 (Knowledge Base)**：預先準備好大量相關文件、資料或歷史對話紀錄。
2. **索引 (Indexing)**：將知識庫中的文本轉換為向量嵌入 (Embeddings - 文本的數字表示)，並存儲在向量資料庫 (Vector Database) 中，以便快速進行語意相似度搜索。常見的向量資料庫有 FAISS, ChromaDB, Pinecone, Weaviate 等。
3. **檢索 (Retrieval)**：當使用者提出問題時： a. 將使用者問題也轉換為向量嵌入。 b. 在向量資料庫中搜索與問題向量最相似的文本片段 (chunks)。
4. **增強 (Augmentation)**：將檢索到的相關文本片段作為額外上下文，與原始問題一起提供給 LLM。
5. **生成 (Generation)**：LLM 基於原始問題和增強的上下文來生成回答。

簡化版 RAG 概念性範例程式碼 (不含真實向量資料庫與嵌入)：(真實 RAG 系統需要 `openai` 產生嵌入，以及如 `faiss-cpu`, `chromadb` 等向量資料庫套件)

```
# 假設 chat_with_gpt 函式已定義

# 0. 模擬知識庫 (實際應用中會是大量文件)
knowledge_base_documents = {
    "doc_ai_agent": "AI Agent 是一種能夠感知環境、進行決策並採取行動以達成特定目標的智慧實體。它通常具備學習能力。",
    "doc_llm": "大型語言模型 (LLM) 是一種深度學習模型，透過在大量文本資料上進行訓練，
```

```
從而理解和生成人類語言。",
    "doc_rag_concept": "檢索增強生成（RAG）是一種結合了資訊檢索系統與大型語言模型的技術。它首先從知識庫中檢索相關資訊，然後將這些資訊提供給 LLM 作為上下文，以生成更準確、更具體的回答。",
    "doc_weather_api": "若要查詢天氣，可以呼叫天氣 API，例如 OpenWeatherMap API，並提供城市名稱作為參數。"
}

# 1. 模擬嵌入與向量資料庫（極簡化，僅用關鍵字匹配）
def retrieve_relevant_documents(query, documents, top_k=2):
    relevant_docs = []
    query_lower = query.lower()
    for doc_id, content in documents.items():
        if any(keyword in content.lower() for keyword in query_lower.split() if len(keyword) > 2): # 簡單關鍵字匹配
            relevant_docs.append(content)
    return relevant_docs[:top_k] # 取回最相關的 top_k 個

# 2. RAG 流程
def answer_with_rag(user_query, conversation_history=None):
    # a. 檢索
    retrieved_texts = retrieve_relevant_documents(user_query, knowledge_base_documents)

    if not retrieved_texts:
        # 如果沒有檢索到相關文件，直接使用 LLM 回答（或給出預設回覆）
        # print("(沒有從知識庫找到特別相關的內容，直接詢問 LLM)")
        return chat_with_gpt(user_query, conversation_history)

    # b. 增強 Prompt
    context_for_llm = "
以下是一些可能相關的背景資訊：
"
    for i, text in enumerate(retrieved_texts):
        context_for_llm += f"[{i+1}] {text}
"

    augmented_prompt = f"請根據以下背景資訊（如果相關）以及你的通用知識來回答使用者的問題。
{context_for_llm}
使用者問題：{user_query}"



    # c. 生成
    # print(f"增強後的 Prompt（部分內容）：{augmented_prompt[:300]}...") # 打印部分 prompt 供參考
    return chat_with_gpt(augmented_prompt, conversation_history)

# 範例使用 RAG
# rag_conversation_history = []
# user_q1 = "什麼是 RAG？它跟 LLM 有什麼關係？"
# answer_rag1, rag_conversation_history = answer_with_rag(user_q1,
```

```
rag_conversation_history)
# print(f"AI (RAG): {answer_rag1}")

# user_q2 = "AI Agent 如何查詢天氣？"
# answer_rag2, rag_conversation_history = answer_with_rag(user_q2,
rag_conversation_history)
# print(f"AI (RAG): {answer_rag2}")
```

---

6. 🤖 AI 小助理開發

小助理概念與功能

AI 小助理是指能協助使用者完成日常任務的 AI Agent，例如行事曆管理、提醒、資料查詢等。

實作方法與範例程式碼

以 Python 建立一個更完整的代辦事項管理，包含新增、檢視、標記完成、移除等功能：

```
class TodoManager:
    def __init__(self):
        self.todos = [] # 每個 todo 可以是 {"task": "內容", "done": False}

    def add_task(self, task_description):
        if not task_description:
            print("任務描述不可為空。")
            return
        self.todos.append({"task": task_description, "done": False})
        print(f"已加入代辦事項: '{task_description}'")

    def view_tasks(self, view_all=True):
        if not self.todos:
            print("目前沒有代辦事項。")
            return

        print("
--- 代辦事項 ---")
        displayed_tasks = False
        for idx, item in enumerate(self.todos):
            if view_all or not item["done"]:
                status = "✅" if item["done"] else "📝"
                print(f"{idx + 1}. {status} {item['task']}")
                displayed_tasks = True
        if not displayed_tasks and not view_all:
            print("所有任務皆已完成！")
        print("----------------")

    def mark_as_done(self, task_number):
        try:
            idx = int(task_number) - 1
            if 0 <= idx < len(self.todos):
```

```
                    if not self.todos[idx]["done"]:
                        self.todos[idx]["done"] = True
                        print(f"任務 '{self.todos[idx]['task']}' 已標記為完成。")
                    else:
                        print(f"任務 '{self.todos[idx]['task']}' 先前已完成。")
                else:
                    print("無效的任務編號。")
            except ValueError:
                print("請輸入有效的數字編號。")

        def remove_task(self, task_number):
            try:
                idx = int(task_number) - 1
                if 0 <= idx < len(self.todos):
                    removed_task = self.todos.pop(idx)
                    print(f"任務 '{removed_task['task']}' 已移除。")
                else:
                    print("無效的任務編號。")
            except ValueError:
                print("請輸入有效的數字編號。")

    # 範例使用 TodoManager
    # my_todo_manager = TodoManager()
    # my_todo_manager.add_task("完成 AI 課程作業")
    # my_todo_manager.add_task("閱讀 LLM 相關論文")
    # my_todo_manager.add_task("運動 30 分鐘")
    # my_todo_manager.view_tasks()
    # my_todo_manager.mark_as_done("1")
    # my_todo_manager.view_tasks(view_all=False) # 只看未完成
    # my_todo_manager.remove_task("3")
    # my_todo_manager.view_tasks()
```

Web API 串接、提醒與通知

Web API（Application Programming Interface）是指網路服務提供的資料交換介面。AI 小助理可透過 API 取得天氣、行事曆等資訊，並主動提醒或通知使用者。

- **API**：應用程式介面，讓不同軟體系統互相溝通。
- **提醒（Reminder）**：主動通知使用者重要事項。
- **通知（Notification）**：即時傳遞訊息給使用者。

實作方法與範例程式碼

範例1：以 Python `requests` 套件取得天氣資訊 (wttr.in) (需要安裝 `requests` 套件: `pip install requests`)

```
    import requests

    def get_weather_simple(city):
        # wttr.in 提供簡單的文字介面天氣查詢
```

```
    url = f"https://wttr.in/{city}?format=3" # format=3 提供簡潔輸出
    try:
        response = requests.get(url, timeout=5) # 設定超時
        response.raise_for_status() # 如果 HTTP 請求返回不成功的狀態碼，則拋出
HTTPError 異常
        print(f"--- {city} 天氣資訊 ---")
        print(response.text.strip())
        print("------------------------")
    except requests.exceptions.RequestException as e:
        print(f"查詢 {city} 天氣時發生錯誤: {e}")

# get_weather_simple("Taipei")
# get_weather_simple("London")
```

範例2：串接 JSONPlaceholder API 取得模擬資料 JSONPlaceholder 是一個提供免費假 REST API 的服務，常用於測試和原型製作。

```
import requests
import json # 用於美化打印 JSON 內容

def get_placeholder_todos(user_id=1):
    api_url = f"https://jsonplaceholder.typicode.com/todos"
    params = {"userId": user_id} # 查詢特定使用者的 todos
    try:
        response = requests.get(api_url, params=params, timeout=5)
        response.raise_for_status()
        todos = response.json() # 將回應內容解析為 Python 字典/列表

        print(f"
--- User {user_id} 的前 3 個 Todos (來自 JSONPlaceholder) ---")
        for todo in todos[:3]: # 只顯示前三個以保持簡潔
            print(f"  ID: {todo['id']}, Title: {todo['title']}, Completed:
{todo['completed']}")
        print("-------------------------------------------------------")
        return todos
    except requests.exceptions.RequestException as e:
        print(f"從 JSONPlaceholder 取得資料時發生錯誤: {e}")
        return None

# get_placeholder_todos(1)
# get_placeholder_todos(2)
```

提醒與通知機制概念

- **排程任務**：使用如 Python 的 `schedule` 套件 (`pip install schedule`) 或作業系統的 `cron` (Linux/macOS) / 工作排程器 (Windows) 來定期執行檢查或發送通知的腳本。
- **推播通知**：若為 Web/App 應用，可整合 Firebase Cloud Messaging (FCM), Apple Push Notification service (APNs) 等服務。
- **郵件通知**：使用 Python 的 `smtplib` 和 `email` 模組發送郵件。
- **通訊軟體整合**：透過 Line Bot, Slack API, Telegram Bot API 等發送訊息。

Python `schedule` 簡易範例：

```
import schedule
import time
import datetime

def daily_reminder():
    print(f"每日提醒 ({datetime.datetime.now()}): 別忘了檢查您的 AI 小助理待辦事項！")

def weekly_report_job():
    print(f"每週報告任務 ({datetime.datetime.now()}): 生成 AI 小助理使用報告...")

# 設定排程
# schedule.every().day.at("09:00").do(daily_reminder) # 每天早上9點執行
# schedule.every().monday.at("08:00").do(weekly_report_job) # 每週一早上8點
# schedule.every(10).minutes.do(daily_reminder) # 每10分鐘執行一次 (用於測試)

# print("排程已設定。腳本將在背景執行排程任務 (此範例僅為演示，不會持續執行)。")
# print("在實際應用中，您需要一個持續運行的迴圈：")
# print("# while True:")
# print("#     schedule.run_pending()")
# print("#     time.sleep(1)")

# 為了在腳本結束前能看到效果，可以手動執行一次 (如果時間到了)
# schedule.run_pending()
# time.sleep(1) # 等待可能的輸出
```

資料收集與 PPT 產生模組

AI 小助理可協助自動收集資料，並產生簡報（PPT, PowerPoint Presentation）檔案，提升工作效率。

- **簡報（PPT）**：用於展示資訊的投影片文件。

實作方法與範例程式碼

資料收集範例：簡易網頁標題爬取 (需要安裝 `requests` 和 `beautifulsoup4` 套件: `pip install requests beautifulsoup4`)

```
import requests
from bs4 import BeautifulSoup

def scrape_website_title(url):
    try:
        headers = { # 偽裝成瀏覽器，有些網站會阻擋爬蟲
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/537.36'
        }
```

```
        response = requests.get(url, headers=headers, timeout=10)
        response.raise_for_status() # 檢查請求是否成功

        # 使用 BeautifulSoup 解析 HTML
        soup = BeautifulSoup(response.text, 'html.parser')

        if soup.title and soup.title.string:
            title = soup.title.string.strip()
            print(f"網站 '{url}' 的標題是: '{title}'")
            return title
        else:
            print(f"在 '{url}' 上找不到標題。")
            return None
    except requests.exceptions.RequestException as e:
        print(f"爬取網站 '{url}' 時發生錯誤: {e}")
        return None
    except Exception as e:
        print(f"處理網站 '{url}' 時發生未知錯誤: {e}")
        return None

# scrape_website_title("https://www.google.com")
# scrape_website_title("https://developer.mozilla.org/zh-
TW/docs/Web/HTML")
```

注意：網頁爬取需遵守目標網站的 `robots.txt` 規範及使用條款，並避免過於頻繁的請求。

PPT 產生範例：使用 `python-pptx` 套件 (需要安裝 `python-pptx` 套件: `pip install python-pptx`)

```
from pptx import Presentation
from pptx.util import Inches, Pt # 用於設定尺寸和字體大小

def create_simple_presentation(filename="ai_summary_report.pptx",
title_text="AI 專案摘要", content_points=None):
    prs = Presentation()

    # 新增標題投影片
    title_slide_layout = prs.slide_layouts[0] # 標題投影片版面
    slide = prs.slides.add_slide(title_slide_layout)
    title_shape = slide.shapes.title
    subtitle_shape = slide.placeholders[1] # 通常是副標題

    title_shape.text = title_text
    subtitle_shape.text = f"自動生成於 {datetime.date.today().strftime('%Y-
%m-%d')}"

    # 新增內容投影片 (如果提供了內容點)
    if content_points and isinstance(content_points, list):
        bullet_slide_layout = prs.slide_layouts[1] # 標題與內容版面
        content_slide = prs.slides.add_slide(bullet_slide_layout)

        title_shape_content = content_slide.shapes.title
        title_shape_content.text = "主要內容點"
```

```
        body_shape = content_slide.shapes.placeholders[1] # 內容區塊
        tf = body_shape.text_frame
        tf.clear() # 清除預設文字

        for point in content_points:
            p = tf.add_paragraph()
            p.text = point
            p.font.size = Pt(18) # 設定字體大小
            p.level = 0 # 設定為第一層項目符號

    try:
        prs.save(filename)
        print(f"簡報檔案 '{filename}' 已成功產生。")
    except Exception as e:
        print(f"儲存簡報檔案 '{filename}' 時發生錯誤: {e}")

# 範例資料
# report_points = [
#     "AI Agent 專案進度符合預期。",
#     "LLM 整合測試已完成初步驗證。",
#     "使用者回饋顯示對新功能反應正面。",
#     "下一步將進行效能優化與壓力測試。"
# ]
# create_simple_presentation(content_points=report_points)
```

---

7. 💬 AI 客服系統設計

客服案例分析

分析真實客服案例，了解常見問題與解決流程。

實作方法

1. **收集與分析真實客服案例**：
   - 來源：公司內部客服對話紀錄、公開的客服案例分享、競品分析等。
   - 分析重點：常見問題類型、使用者意圖、解決方案、對話流程、情緒轉折點。
   - 建立客戶畫像 (Customer Persona) 和客戶旅程地圖 (Customer Journey Map) 來理解不同類型客戶的需求與痛點。
2. **定義 AI 客服的範圍與目標**：
   - AI 客服主要處理哪些問題？（例如：常見問題解答、訂單查詢、技術支援初步排查）
   - AI 客服的目標是什麼？（例如：提升首次問題解決率、縮短平均處理時間、提升客戶滿意度）
   - 何時需要轉接人工客服？定義清晰的轉接條件與流程。

拆解客服對話與會話紀錄設計

會話紀錄（Conversation Log）是指保存使用者與客服互動內容，便於追蹤與分析。

- **會話（Conversation）**：雙方互動的訊息交換過程。

- **紀錄（Log）**：保存事件或資料的檔案。

實作方法與範例程式碼

以 Python 列表與字典儲存結構化的會話紀錄：

```
import datetime

class ConversationLogger:
    def __init__(self, conversation_id):
        self.conversation_id = conversation_id
        self.log = [] # 每條紀錄包含 'speaker', 'text', 'timestamp',
'metadata' (可選)

    def add_entry(self, speaker, text, metadata=None):
        """
        新增一條對話紀錄。
        :param speaker: 'user' 或 'ai' (或更具體的客服/使用者名稱)
        :param text: 對話內容
        :param metadata: (可選) 額外資訊，如意圖、情感分析結果等
        """
        entry = {
            "speaker": speaker,
            "text": text,
            "timestamp": datetime.datetime.now().isoformat(), # ISO 格式時間
戳
            "metadata": metadata if metadata else {}
        }
        self.log.append(entry)
        # print(f"Log added for {self.conversation_id}: {speaker} -
{text[:30]}...") # 打印部分日誌

    def get_log(self):
        return self.log

    def print_formatted_log(self):
        print(f"
--- 會話紀錄 (ID: {self.conversation_id}) ---")
        if not self.log:
            print(" (此會話尚無紀錄)")
            return
        for entry in self.log:
            meta_info = f" (Meta: {entry['metadata']})" if
entry['metadata'] else ""
            print(f"[{entry['timestamp']}] {entry['speaker']}:
{entry['text']}{meta_info}")
        print("-----------------------------------------")

# 範例使用
# conv_id = "CONV_001"
# logger = ConversationLogger(conv_id)
```

```
# logger.add_entry("user", "你好，我想查詢我的訂單狀態。")
# logger.add_entry("ai", "好的，請您提供您的訂單編號。", metadata={"intent":
"request_order_number"})
# logger.add_entry("user", "訂單編號是 A12345678。")
# logger.add_entry("ai", "感謝您，正在為您查詢訂單 A12345678 的狀態...",
metadata={"entities": {"order_id": "A12345678"}})
# logger.print_formatted_log()

# all_logs_for_conv_001 = logger.get_log()
# print(f"
原始日誌資料 for {conv_id}: {all_logs_for_conv_001}")
```

會話紀錄設計考量：

- **唯一識別碼**：為每次會話分配唯一 ID。
- **參與者**：明確是使用者還是 AI (或特定客服)。
- **時間戳**：記錄每條訊息的精確時間。
- **訊息內容**：原始文本。
- **元數據 (Metadata)**：
  - **意圖 (Intent)**：使用者這句話想做什麼 (例如：查詢訂單、詢問產品)。
  - **實體 (Entities)**：訊息中的關鍵資訊 (例如：訂單號、產品名稱、地點)。
  - **情感分析 (Sentiment)**：使用者情緒 (正面/負面/中性)。
  - **AI 回應的信心度**。
  - **是否轉接人工**。
- **儲存格式**：JSON, CSV, 或存入資料庫 (如 MongoDB, PostgreSQL)。

下單處理與訂單結算模擬

下單（Order Placement）是指顧客提交購買需求，**訂單結算（Order Settlement）**則是計算金額、完成交易的過程。

- **結算（Settlement）**：完成交易並計算金額。

實作方法與範例程式碼

```
class OrderManager:
    def __init__(self):
        self.orders = [] # 儲存訂單的列表
        self.next_order_id = 1 # 自動產生訂單 ID

    def place_order(self, customer_id, items, payment_method):
        """
        處理下單請求。
        :param customer_id: 顧客 ID
        :param items: 一個包含商品字典的列表，例如 [{"item_name": "珍珠奶茶",
"quantity": 1, "price_per_unit": 60}, ...]
        :param payment_method: 付款方式，例如 "信用卡", "貨到付款"
        :return: 訂單 ID，如果成功；否則返回 None
        """
```

```
        if not customer_id or not items:
            print("錯誤：顧客 ID 和商品列表不可為空。")
            return None

        order_id = f"ORD{self.next_order_id:04d}" # 例如 ORD0001
        self.next_order_id += 1

        total_amount = sum(item["quantity"] * item["price_per_unit"] for
item in items)

        order_details = {
            "order_id": order_id,
            "customer_id": customer_id,
            "items": items,
            "total_amount": total_amount,
            "payment_method": payment_method,
            "status": "pending_payment", # 初始狀態：等待付款
            "timestamp": datetime.datetime.now().isoformat()
        }
        self.orders.append(order_details)
        print(f"訂單 #{order_id} 已建立 (顧客: {customer_id}, 金額:
{total_amount}, 狀態: {order_details['status']})。")
        return order_id

    def update_order_status(self, order_id, new_status):
        for order in self.orders:
            if order["order_id"] == order_id:
                old_status = order["status"]
                order["status"] = new_status
                print(f"訂單 #{order_id} 狀態已從 '{old_status}' 更新為
'{new_status}'。")
                # 在此處可以觸發通知等後續操作
                return True
        print(f"錯誤：找不到訂單 #{order_id}。")
        return False

    def get_order_details(self, order_id):
        for order in self.orders:
            if order["order_id"] == order_id:
                return order
        return None

    def settle_payment(self, order_id, payment_confirmation_code):
        # 模擬結算付款
        if self.update_order_status(order_id, "payment_confirmed"):
            print(f"訂單 #{order_id} 付款已確認 (確認碼:
{payment_confirmation_code})。準備出貨。")
            self.update_order_status(order_id, "processing") # 更新狀態為處理
中
            return True
        return False

    def view_all_orders(self):
        if not self.orders:
```

```
            print("目前沒有任何訂單。")
            return
        print("
--- 所有訂單列表 ---")
        for order in self.orders:
            print(f"  ID: {order['order_id']}, 顧客:
{order['customer_id']}, 金額: {order['total_amount']}, 狀態:
{order['status']}")
        print("--------------------")

# 範例使用 OrderManager
# order_system = OrderManager()

# items1 = [{"item_name": "珍珠奶茶", "quantity": 2, "price_per_unit": 60},
{"item_name": "雞排", "quantity": 1, "price_per_unit": 80}]
# order1_id = order_system.place_order("CUST001", items1, "信用卡")

# items2 = [{"item_name": "經典紅茶", "quantity": 1, "price_per_unit": 30}]
# order2_id = order_system.place_order("CUST002", items2, "悠遊卡")

# if order1_id:
#     order_system.settle_payment(order1_id, "PAY_CONF_XYZ123")
#     order_system.update_order_status(order1_id, "shipped")

# order_system.view_all_orders()

# details = order_system.get_order_details(order2_id)
# if details:
#     print(f"
訂單 {order2_id} 詳細資料: {details}")
```

通知用戶下訂成功、安排排單邏輯

系統需主動通知用戶下訂成功，並根據規則安排處理順序（排單邏輯，Queue Logic）。

- **排單（Queue）**：依序處理多個任務的機制。

實作方法與範例程式碼

通知用戶：

- **方式**：電子郵件、簡訊 (SMS)、App 推播通知、通訊軟體訊息 (如 Line, WhatsApp)。
- **內容**：訂單編號、訂單摘要、預計送達時間、感謝訊息等。
- **時機**：下訂成功時、付款確認時、出貨時、送達時等。

排單邏輯 (Queue Logic)： 使用 Python 的 `collections.deque` 可以方便地實現先進先出 (FIFO) 的佇列。

```
from collections import deque
import time # 用於模擬處理時間
```

```
class OrderProcessingQueue:
    def __init__(self):
        self.queue = deque() # 使用 deque 作為佇列

    def add_order_to_queue(self, order_id):
        self.queue.append(order_id)
        print(f"訂單 #{order_id} 已加入處理佇列。目前佇列長度:
{len(self.queue)}")

    def process_next_order(self):
        if not self.queue:
            print("佇列中沒有待處理的訂單。")
            return None

        order_id_to_process = self.queue.popleft() # 從佇列左側取出 (先進先出)
        print(f"開始處理訂單 #{order_id_to_process}...")
        # --- 模擬訂單處理過程 ---
        # 例如：檢查庫存、包裝商品、安排物流等
        time.sleep(2) # 模擬處理耗時 2 秒
        # --- 處理完成 ---
        print(f"訂單 #{order_id_to_process} 已處理完成。")
        # 在此處可以觸發「已出貨」的通知
        return order_id_to_process

    def view_queue(self):
        print(f"目前待處理佇列: {list(self.queue)}")

# 範例使用 OrderProcessingQueue
# (假設 order_system 和 order1_id, order2_id 已如前述範例定義並成功建立訂單)
# processing_queue = OrderProcessingQueue()

# if order1_id and order_system.get_order_details(order1_id)['status'] ==
'processing': # 假設付款成功後狀態為 processing
#     processing_queue.add_order_to_queue(order1_id)
# if order2_id and order_system.get_order_details(order2_id)['status'] ==
'pending_payment': # 假設 order2 尚未付款，先不加入佇列
#     print(f"訂單 {order2_id} 尚未付款，暫不加入處理佇列。")
#     # 假設 CUST002 完成付款
#     # order_system.settle_payment(order2_id, "PAY_CONF_ABC789")
#     # if order_system.get_order_details(order2_id)['status'] ==
'processing':
#     #     processing_queue.add_order_to_queue(order2_id)


# processing_queue.view_queue()
# processing_queue.process_next_order()
# processing_queue.view_queue()
# processing_queue.process_next_order()
# processing_queue.process_next_order() # 佇列已空
```

更進階的排單邏輯：

- **優先級佇列 (Priority Queue)**：例如 VIP 客戶的訂單優先處理。可以使用 Python 的 `heapq` 模組實現。

- **基於資源的排程**：考慮到處理能力（例如：廚房每小時能做多少杯飲料）。
- **分組處理**：將相似的訂單（例如：同一外送區域）集中處理。

---

8. 🧪 期末整合與測試

系統部署與上線

部署（Deployment）是指將開發完成的系統安裝到正式運作環境，讓使用者可以存取。

- **上線（Go Live）**：系統正式對外開放使用。

實作方法

1. **選擇部署環境**：
   - **雲端平台**：AWS (EC2, Elastic Beanstalk, Lambda), Google Cloud Platform (Compute Engine, App Engine, Cloud Run), Microsoft Azure (VMs, App Service), Heroku, DigitalOcean, Vercel (尤其適合前端與 Next.js), Railway 等。
   - **自建伺服器 (On-premise)**：需要自行管理硬體與網路。
2. **容器化 (Containerization) - 推薦**：
   - 使用 Docker 將應用程式及其所有依賴打包成一個標準化的容器映像檔 (Image)。
   - **優點**：環境一致性（開發、測試、生產環境相同）、易於擴展、快速部署。
   - `Dockerfile` **範例 (Python Flask/FastAPI 應用)**：

```
# 使用官方 Python 映像檔作為基礎
FROM python:3.9-slim-buster

# 設定工作目錄
WORKDIR /app

# 複製依賴需求文件並安裝（分開複製以利用 Docker 快取）
COPY requirements.txt requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# 複製應用程式的其餘程式碼
COPY . .

# 設定環境變數（可選，例如 API 金鑰，但更推薦使用雲平台的秘密管理）
# ENV
OPENAI_API_KEY="your_actual_api_key_here_or_read_from_secret_mana
ger"
# ENV FLASK_APP=your_main_app_file.py # 如果是 Flask

# 開放應用程式運行的端口（例如 Flask/FastAPI 預設 5000/8000）
EXPOSE 8000

# 啟動應用程式的命令（以 FastAPI/uvicorn 為例）
# CMD ["uvicorn", "your_main_app_file:app", "--host", "0.0.0.0",
"--port", "8000"]
# 若為 Flask:
# CMD ["flask", "run", "--host=0.0.0.0", "--port=8000"]
```

3. **設定 CI/CD (持續整合/持續部署) 管線**：
    - 工具：GitHub Actions, GitLab CI/CD, Jenkins, CircleCI。
    - 流程：程式碼提交到版本控制 (如 Git) -> 自動觸發測試 -> 自動建置容器映像檔 -> 自動部署到測試/生產環境。
4. **環境變數與組態管理**：
    - 將 API 金鑰、資料庫連線字串等敏感資訊或環境特定設定，透過環境變數注入，而非硬編碼在程式中。
    - 雲平台通常提供秘密管理服務 (如 AWS Secrets Manager, GCP Secret Manager)。
5. **資料庫與儲存設定**：
    - 選擇並設定適合的資料庫 (如 PostgreSQL, MySQL, MongoDB)。
    - 設定檔案儲存 (如 AWS S3, Google Cloud Storage)。
6. **網域名稱與 SSL/TLS 憑證**：
    - 設定網域名稱指向您的應用程式。
    - 為 HTTPS 設定 SSL/TLS 憑證 (例如使用 Let's Encrypt 免費憑證)。
7. **監控與日誌**：
    - 部署後，設定監控工具 (如 Prometheus, Grafana, Datadog, Sentry, 或雲平台內建監控) 來追蹤系統效能、錯誤率、資源使用情況。
    - 集中管理應用程式日誌。

整合測試與效能調整

整合測試（Integration Testing）是指檢查多個模組協同運作是否正常。**效能調整（Performance Tuning）**則是針對系統速度、穩定性進行優化。

- **效能（Performance）**：系統運作的速度與效率。
- **調整（Tuning）**：針對特定目標進行優化。

實作方法

1. **撰寫整合測試案例**：
    - 測試不同模組間的互動是否如預期。例如：使用者輸入 -> NLU 模組 -> 意圖識別 -> 業務邏輯模組 -> LLM 呼叫 -> 回應生成 -> 使用者介面。
    - 涵蓋主要的使用者流程與邊界條件。
    - 可以使用測試框架如 `pytest` (Python)。
2. **執行整合測試**：
    - 在類生產環境中執行測試。
    - CI/CD 流程中應包含自動化整合測試步驟。
3. **效能測試與分析**：
    - **負載測試 (Load Testing)**：模擬多使用者同時存取，觀察系統在高負載下的回應時間、吞吐量、錯誤率。工具如 Locust, k6, Apache JMeter。
    - **壓力測試 (Stress Testing)**：測試系統在極限負載下的穩定性與恢復能力。
    - **瓶頸分析 (Bottleneck Analysis)**：找出系統效能瓶頸所在（例如：CPU、記憶體、網路、資料庫查詢、外部 API 呼叫）。
        - 使用程式碼剖析工具 (Profiler)。
        - 檢查 LLM API 呼叫的延遲。

4. **效能調整策略**：
    - **LLM 相關優化**：
        - **提示詞優化 (Prompt Optimization)**：更簡潔、明確的提示詞可能減少 LLM 處理時間與成本。
        - **模型選擇 (Model Selection)**：根據需求選擇最適合的模型。例如，對於簡單任務，不一定需要最強大（也最慢、最貴）的模型。
        - **快取 (Caching)**：對於重複性高且不常變動的查詢，可以快取 LLM 的回應。
        - **批次處理 (Batching)**：如果 LLM API 支援，將多個請求合併為一個批次發送，可能提升效率。
        - **串流輸出 (Streaming)**：對於生成較長文本的場景，逐步串流輸出結果，可以改善使用者的感知效能，讓使用者更快看到部分內容。
        - **減少 Token 使用**：優化輸入輸出的長度，降低成本與延遲。
    - **應用程式層級優化**：
        - **非同步處理 (Asynchronous Operations)**：對於 I/O 密集型操作（如呼叫外部 API、資料庫查詢），使用非同步程式設計（如 Python 的 `asyncio`）避免阻塞主執行緒。
        - **資料庫優化**：優化查詢語句、建立索引、使用連接池。
        - **程式碼優化**：改進演算法效率、減少不必要的計算。
        - **水平擴展 (Horizontal Scaling)**：增加應用程式實例數量。
        - **負載平衡 (Load Balancing)**：將請求分散到多個應用程式實例。
5. **持續監控與迭代**：上線後持續監控系統效能，並根據實際運行情況進行調整。

✅ 學習評估方式

- **反應評估**：課後滿意度調查，了解學員對課程的即時反饋。
- **學習評估**：期末報告與簡報，檢視學員學習成果。
- **行為評估**：課後實作行動計畫，觀察學員實際應用能力。

> 本教學文件針對 AI Agent 與相關專有名詞進行詳細說明，協助學員建立完整知識架構，為後續實作打下堅實基礎。