多輪引導漏斗框架：目標導向對話式AI的架構藍圖

第一部分：引導漏斗的概念架構

對話式AI從簡單的反應式聊天機器人發展為複雜的目標導向代理，這代表了人機交互的典範轉移。對於商業應用，特別是在銷售和客戶服務領域，目標不再僅僅是回答問題，而是主動引導用戶向特定且有價值的結果前進。這需要一個能夠管理結構化多輪對話的架構，以類似傳統銷售漏斗的方式逐步縮小對話範圍。本節為這樣的系統建立理論基礎，將以商業為中心的銷售漏斗概念轉換為強健的對話式AI架構。它解構了問題並將其映射到對話系統設計中已建立和新興的模式，為實用且強大的實現奠定基礎。

1.1節：從銷售漏斗到對話流程：新典範

「對話漏斗」的核心概念是將傳統營銷和銷售漏斗直接轉換到自動化對話領域。[1]在經典銷售模型中，潛在客戶會經歷不同階段——從初始認知到最終購買和保留。[2]對話漏斗旨在在聊天介面的限制內複製這個旅程，將AI代理從被動的資訊儲存庫轉變為主動的策略指南。這種方法基於這樣的理解：自動化對話不應該是漫無目的的，而必須與商業目標整合並由其驅動。

典型銷售漏斗的階段為對話流程提供了清晰的藍圖。這些階段包括認知、興趣、評估、參與、行動和保留。[2]精心設計的對話代理可以系統性地引導用戶經歷這些階段。例如，初始互動可以通過歡迎用戶並介紹品牌目的來建立認知，就像人類代理會做的那樣。[3]隨著對話的進展，代理可以透過提出針對性問題來發掘用戶需求和痛點，從而引起興趣。在評估階段，代理提供具體資訊，比較選項，並幫助用戶評估產品的適合性。接下來是參與階段，代理可能提供演示或特殊促銷。行動階段是漏斗的高潮，代理直接在聊天視窗內促進購買或註冊過程。最後，購買後的互動可以透過收集回饋或提供支援來促進保留。[2]

這種漏斗的實施帶來顯著的商業優勢，包括提高潛在客戶轉換率、改善銷售效率和更高的客戶參與度。[1]透過自動化資格審查和培育過程，銷售團隊可以專注於高價值活動，導致更快的交易完成和大幅的成本節約。[1]

這個模型需要對話代理設計理念的根本轉變——從反應式協助轉向主動引導。在許多商業環境中常見的標準Retrieval-Augmented Generation (RAG)應用採用「拉取」模型；它們等待用戶查詢並檢索相關答案。用戶完全控制對話方向。然而，「引導對話走向真正需求」的商業要求需要「推送」模型，代理主動將對話導向預定目標。對話漏斗的研究一致強調這種主動性，使用諸如「引導潛在客戶」、「發起互動」和「培育潛在客戶」等動詞。[1]簡單

的輸入 -> LLM -> 輸出循環在架構上對這項任務是不足的。它缺乏維持持續目標、制定逐輪策略和根據用戶反應調整方法的機制。因此，需要更複雜的架構，包含強健的狀態管理和深思熟慮的多步推理過程，以有效執行對話漏斗。

1.2節：Dialogue Manager：系統的認知核心

任何先進的任務導向對話系統的核心都是Dialogue Manager (DM)。DM是負責管理狀態和控制對話流程的中央元件，作為系統的認知核心。[5]它協調整個互動，接收語意輸入，與外部知識庫（如產品目錄）介接，追蹤對話進展，並決定系統的下一個行動。[5]DM的職責可以分解為兩個主要的相互關聯任務：Dialogue State Tracking (DST)和Dialogue Control。

Dialogue State Tracking (DST)是在任何給定時刻維護對話狀態的結構化表示的過程。[7]這個狀態包括用戶意圖、已收集的具體資訊片段（實體或slots）以及對話歷史。[7]實質上，DST是DM的記憶。它允許系統理解當前語境與之前一切的關係，這對生成連貫且相關的回應是必要的。[7]對我們的對話漏斗來說，狀態會追蹤用戶處於哪個漏斗階段以及目前為止從他們那裡收集了什麼資訊。

Dialogue Control是DM的決策能力。基於當前對話狀態，控制機制決定系統應該採取的下一個行動。[5]這可能是提出澄清問題、提供建議、查詢資料庫，甚至將對話升級給人類代理。在引導漏斗中，對話控制政策至關重要；它體現了推動對話前進的商業邏輯。

歷史上，DM通常實現為嚴格的基於規則的系統，如Finite-State Machines (FSMs)，對話只能沿著預定義的路徑進行。[8]雖然可預測，但這些系統很脆弱，難以處理人類語言的自然變異性。Large Language Models (LLMs)的出現允許新方法：使用LLM本身作為靈活的機率Dialogue Manager。然而，在單一的整體prompt中委託LLM所有DM功能可能不可靠且不透明，特別是當必須嚴格執行複雜商業邏輯時。[9]LLM可能會偏離軌道、忘記關鍵步驟，或「幻想」出偏離預期漏斗的對話路徑。

更強健的架構模式將DM的核心功能分離為不同的LLM驅動步驟。這是用戶要求的兩層prompt系統的基礎。這個「Think, Then Act」循環是DM傳統「Track State -> Decide Control -> Generate Response」循環的直接LLM原生實現。

1. **Dialogue Control（"Think"步驟）**：第一個LLM調用（prompt-lvl1）充當Dialogue Control模組。它接收當前對話狀態（歷史、收集的資訊）和最新的用戶輸入。其唯一目的是分析這個語境並決定推動漏斗前進所需的下一個策略行動。輸出不是面向用戶的句子，而是結構化命令（例如，特定指令或JSON物件）。這是作為推理引擎運行的LLM調用。

2. **Natural Language Generation（"Act"步驟）**：第二個LLM調用（prompt-lvl2）充當Natural Language Generation (NLG)模組。它接收「Think」步驟決定的策略行動，並將其轉換為優雅的、語境適當的、面向用戶的回應。
3. **Dialogue State Tracking（框架的角色）**：將使用LangGraph等工具實現的總體應用框架負責DST。它明確地儲存、更新並在轉換間傳遞DialogueState物件，確保「Think」步驟始終具有完美的最新對話記憶。

通過這樣構建系統，我們不僅僅是「prompting an LLM」。我們正在架構一個由LLM驅動的Dialogue Manager，它利用模型強大的語言和推理能力，同時施加可靠執行目標導向對話漏斗所需的結構和控制。這種方法以LLMs的靈活性取代舊FSMs的僵化，而不犧牲商業應用所需的可預測性。

1.3節：Slot Filling：漏斗進展的引擎

Slot filling是驅動對話漏斗前進的主要機械過程。它是對話式AI中用於從用戶收集特定必需資訊片段以完成任務或意圖的技術。[10]在引導漏斗的語境中，slots是代理在能夠成功提出產品建議之前需要收集的基本數據點的預定義佔位符。[11]每個填滿的slot代表深入漏斗的一步，將用戶從一般查詢移向特定的可行偏好集。

概念上，slots可以被視為函數的參數，其中函數是最終商業目標——例如，recommend_coffee(taste_profile, caffeine_level, brew_method)。[11]如果用戶的初始輸入沒有提供所有必要的「參數」，系統必須主動參與未填充slots澄清的過程。[11]這是代理「引導」本質最明顯的地方。系統查詢其狀態，識別下一個必需但目前空白的slot，並生成針對性問題以從用戶那裡引出那個特定資訊片段。[11]這種逐輪資訊收集系統性地縮小可能結果範圍，確保最終建議高度相關且個人化。

設計強健的slot-filling系統需要清晰的架構來定義每個要收集的資訊片段。每個slot的關鍵參數包括[11]：

- **Name**：slot的唯一識別符（例如，taste_profile）。
- **Entity**：slot期望的資訊類型，通常連結到預定義值列表或更複雜的實體識別器（例如，咖啡風味筆記的自定義實體）。
- **Required**：布林旗標，指示slot是否為完成任務的必要項。這個旗標至關重要，因為它決定了對話控制政策。系統必須繼續澄清過程直到所有必需slots都被填滿。
- **Is Array**：指示slot是否可以接受多個值的旗標（例如，用戶可能同時喜歡「fruity」和「chocolatey」口味特徵）。

雖然強大，slot filling並非沒有挑戰。用戶輸入可能含糊不清（例如，「I like strong coffee」可能指咖啡因水平、烘焙程度或風味強度），理解正確含義往往需要深度語境意識。[12]設計良好的系統必須能夠處理這種歧義，或許透過提出進一步的澄清問題或提供多選選項來消除用戶意圖的歧義。

對於我們的咖啡銷售範例，以下架構提供支撐代理資訊收集過程的決定性數據結構。此表格作為DialogueState的單一真實來源，將「推薦咖啡」的抽象目標轉換為將驅動對話的具體數據需求集。

表格：咖啡銷售Slot架構

Slot名稱	數據類型	漏斗中的目的	必需？	範例澄清問題
taste_profile	List（例如，"fruity", "chocolatey", "nutty"）	縮小風味偏好範圍（興趣/評估階段）	是	"為了幫我為您找到完美的咖啡，您能告訴我您通常喜歡什麼樣的風味嗎？例如，您喜歡fruity、chocolatey，還是更earthy的口味？"
caffeine_level	String（"regular", "decaf", "half-caff"）	確定用戶的咖啡因需求（評估階段）	是	"您在尋找一般咖啡，還是比較偏好decaf？"
brew_method	String（例如，"espresso", "drip", "pour-over", "french press"）	將咖啡豆研磨和類型與用戶設備匹配（評估階段）	否	"您有偏好的沖煮方法嗎，比如espresso或drip coffee maker？這可以幫我縮小選擇範圍。"
budget_per_bag	Integer	確保建議在用戶價格範圍內（評估階段）	否	"您對一袋咖啡有特定的預算考量嗎？"
experience_level	String（"beginner", "intermediate", "expert"）	根據推薦咖啡和沖煮建議的複雜性進行調整（個人化）	否	"為了給出最好的建議，您認為自己是咖啡初學者還是更有經驗的家庭沖煮者？"

1.4節："Think, Then Act"循環：透過Chaining實現兩步Prompting

用戶對兩層prompt系統的核心要求可以正式化為稱為「Think, Then Act」循環的架構模式。這個模式是Prompt Chaining的實際實現，這是一種將複雜任務分解為較小、相互連接的prompts序列的技術，其中一個prompt的輸出作為下一個的輸入。[13]與使用單一的整體prompt相比，這種方法顯著改善了LLM驅動應用的可靠性、可控性和透明度。[15]

「Think, Then Act」循環將每個對話轉換分為兩個不同階段：

1. **"Think"階段（分析和規劃）**：鏈中的第一個prompt（prompt-lvl1）不是設計來生成面向用戶的文本。其目的是對當前DialogueState執行策略分析。它接收對話歷史、當前已填充的slots和用戶最新訊息作為輸入。其任務是決定推動漏斗前進所需的下一個邏輯行動。這利用了Dynamic Prompt Adaptation的概念，系統分析語境以確定當前輪次的特定目標。[16]這個「Think」prompt的輸出是結構化指令——例如，像{"action": "ELICIT_SLOT", "parameter": "taste_profile"}這樣的JSON物件。這步驟純粹用於內部推理。
2. **"Act"階段（回應生成）**：鏈中的第二個prompt（prompt-lvl2）接收「Think」階段生成的結構化指令作為其主要輸入。其任務是將該指令轉換為自然、引人入勝且面向用戶的回應。例如，如果它接收到引出taste_profile slot的指令，它會生成像「我絕對可以幫助您！首先，您通常在咖啡中喜歡什麼樣的風味？」這樣的訊息。這步驟是更標準的專注生成任務。

這個架構是Conditional Chaining或Dynamic Chaining的一種形式。[17]流程不是簡單的線性序列。相反，「Think」步驟的輸出動態決定接下來應該執行哪個「Act」prompt或邏輯。如果「Think」步驟決定行動是RECOMMEND，將觸發與ELICIT_SLOT不同的「Act」prompt和過程。這創造了靈活但受控的對話流程，可以在每個轉換中適應對話的需求。

這種關注點分離不僅僅是實現細節；它是構建強健的目標導向代理的關鍵設計模式。LLMs的主要失敗模式是它們傾向於「幻想」或偏離嚴格的基於規則的邏輯，特別是當給予複雜的多部分指令時。[9]商業漏斗的核心是一組規則（例如，「如果taste_profile缺失，您必須詢問它」）。

嘗試使用單一大型prompt如「您是咖啡銷售代理。與用戶交談，遵循漏斗邏輯，並引導他們進行銷售」來執行這些規則給LLM太多創意自由。它可能忘記提出必需問題，被離題查詢分散注意力，或發明自己的對話路徑，打破漏斗邏輯。

「Think, Then Act」架構透過在每個步驟約束LLM任務來減輕這種風險。

- **"Think" prompt受到嚴重約束**。它的指令不是聊天，而是執行分類任務：「基於對話狀態，從這個特定列表中選擇下一個行動：``。以JSON物件輸出您的選擇。」強制LLM產生像JSON這樣的結構化數據大幅減少其輸出空間並最小化對話漂移或創意幻想的機會。這是一種結構化知識融入形式，奠定模型推理的基礎。[18]
- **"Act" prompt接收更簡單、更專注的任務**。例如：「系統的計劃是為taste_profile ELICIT_SLOT。生成一個友好的問題向用戶詢問這個資訊。」這減少了任何單一生成步驟中模型的認知負載，增加了最終輸出的準確性和可靠性。

最終，這個兩步架構提供了執行漏斗基於狀態的商業邏輯的強大機制。它允許系統利用LLM細緻的語言理解進行決策，同時防止模型偏離核心任務，從而增強對話代理的整體控制和可預測性。[14]

第二部分：使用Python和LangGraph的實用框架實現

將引導漏斗框架的概念架構轉換為功能性應用需要仔細選擇技術棧和有條理的實現過程。本節提供使用Python構建咖啡銷售代理的詳細實用藍圖。它從理論轉向代碼，演示如何構建狀態管理、對話邏輯和動態prompting機制，使「Think, Then Act」循環生效。重點是創建完整的、可工作的系統，作為第一部分討論原則的具體範例。

2.1節：技術藍圖：Python、LangChain和LangGraph

技術選擇對於有效構建和擴展複雜對話代理至關重要。此實現選擇的技術棧是Python、LangChain和LangGraph，每個都因其在開發LLM驅動應用中的特定優勢而被選擇。

- **Python**：作為AI和機器學習開發的事實標準語言，Python提供豐富的函式庫生態系統、廣泛的社群支援和直接的語法，使其成為我們專案的理想基礎。
- **LangChain**：這個開源框架為使用LLMs構建應用提供了全面的工具和抽象集。[19]它透過提供用於prompt管理、模型互動、記憶和數據檢索的模組化元件來簡化許多常見任務。對於我們的代理，我們將利用幾個關鍵的LangChain元件[19]： ○ **Chat Models**：針對對話互動最佳化的各種LLMs（例如來自OpenAI、Anthropic、Google）的介面。 ○ **Prompt Templates**：允許透過插入像聊天歷史和用戶輸入這樣的變數來動態構建prompts的可重複使用模板。 ○ **Memory**：用於儲存和檢索對話歷史的元件，這對維持語境是必要的。
- **LangGraph**：雖然LangChain很適合構建線性操作序列（chains），引導漏斗框架需要更複雜的控制流程。對話不是直線；它是基於對話狀態的具有條件分支的循環。來自LangChain的簡單ConversationChain是不足的，因為它無法輕易容納「Think, Then Act」邏輯，其中必須在每個轉換做出決定以確定下一步。這正是LangGraph設計要解決的問題。[20]LangGraph是LangChain的擴展，允許開發者透過將它們表示為圖形來構建有狀態的多代理應用。其關鍵特性使其成為我們架構的完美工具： ○ **Stateful Graphs**：LangGraph允許定義中央狀態物件（我們的DialogueState），該物件被明確傳遞給圖中的每個節點並由其更新。這提供了Dialogue State Tracking的強健機制。 ○ **Nodes and Edges**：應用邏輯構建為由節點（函數或可執行物件）和邊緣（指導節點間流程）組成的圖。這允許我們為「Think」步驟（router）創建節點，為每個「Act」

步驟（例如，elicit、recommend）創建單獨的節點。 ○ **Conditional Edging**：LangGraph允許動態決定控制流程。我們可以創建條件邊緣，根據我們「Think」（router）節點的輸出將對話路由到不同的「Act」節點。這直接實現了我們框架所需的條件和動態chaining模式。[17] ○ **Cycles**：它自然支援循環，這對對話是基本的。在「Act」節點生成回應後，流程可以循環回來等待下一個用戶輸入，然後重新進入「Think」節點開始下一輪。

總結來說，LangChain為與LLMs互動提供基本構件，而LangGraph提供將這些構件編排為我們引導漏斗代理的複雜、有狀態和循環邏輯所需的關鍵控制流程引擎。

2.2節：定義DialogueState和咖啡架構

在構建對話邏輯之前，必須定義將管理代理記憶和知識的數據結構。這些結構形成系統的骨幹，為追蹤對話和存取產品資訊提供一致格式。

DialogueState

DialogueState是代表代理在對話中任何時點記憶的中央物件。它將在我們的LangGraph應用中的節點間傳遞，確保系統的每個部分都能存取完整的對話語境。使用Python typing函式庫的TypedDict提供類型提示以提升清晰度和開發者體驗。

狀態將包含以下關鍵欄位：

- **chat_history**：BaseMessage物件列表（來自LangChain），儲存對話的逐輪歷史。這對向LLM提供語境至關重要。
- **filled_slots**：儲存從用戶收集資訊的字典。此字典的鍵將對應我們咖啡銷售Slot架構中的Slot名稱（例如，'taste_profile'），值將是用戶提供的資訊。
- **recommendations**：包含代理生成產品建議的字串列表，以避免重複建議。

以下是Python實現：

```python
```

```python
from typing import TypedDict, List, Dict, Any
from langchain_core.messages import BaseMessage

class DialogueState(TypedDict):
    """
    代表對話的狀態。
    """
    chat_history: List
    filled_slots: Dict[str, Any]
    recommendations: List[str]
```

咖啡產品架構和知識庫

代理做出相關建議的能力取決於其產品的結構化知識庫。對於這個例子，我們假設咖啡數據儲存在名為coffee_products.csv的CSV檔案中。此檔案應包含產品名稱、描述、口味筆記、咖啡因水平、適合的沖煮方法、價格等欄位。

這些數據需要在應用啟動時載入。將外部數據載入Documents的概念是LangChain的核心模式，常用於RAG。[19]對於我們的初始實現，我們可以使用Pandas函式庫將CSV載入為更易存取的格式，如字典列表。這將允許在建議邏輯中輕鬆篩選。

python

```python
import pandas as pd

def load_coffee_knowledge_base(filepath: str = "coffee_products.csv") -> List:
    """
    從CSV檔案載入咖啡產品目錄。
    """
    try:
        df = pd.read_csv(filepath)
        # 轉換為字典列表以便存取
        return df.to_dict(orient='records')
    except FileNotFoundError:
        print(f"錯誤：在{filepath}找不到知識庫檔案")
        return

# 應用啟動時載入知識庫
coffee_knowledge_base = load_coffee_knowledge_base()
```

這個結構化的DialogueState和coffee_knowledge_base為構建代理邏輯提供必要基礎。狀態物件將在對話過程中更新，知識庫將在做建議時被查詢。

2.3節：使用LangGraph構建狀態機

本節詳述使用LangGraph構建對話代理核心邏輯。代理被建模為狀態圖，其中節點代表處理步驟（「Think」和「Act」），邊緣指導對話流程。

圖和狀態初始化

首先，我們初始化StatefulGraph。這個圖綁定到我們的DialogueState物件，該物件將傳遞給每個節點並可被修改。

```python
from langgraph.graph import StateGraph, END

# 初始化圖
workflow = StateGraph(DialogueState)
```

Router節點（"Think"步驟）

router是我們圖中最關鍵的節點。它體現我們循環的「Think」部分。它的工作是分析當前DialogueState並決定接下來應該執行哪個「Act」節點。這個決定透過專門的「Router Prompt」指導的LLM調用做出。

router函數將：

1. 接收當前DialogueState。
2. 識別哪些必需slots（來自我們的架構）仍然空白。
3. 為LLM構建prompt，包括對話歷史、已填充slots的狀態和選擇下一個行動的清晰指令。
4. 調用LLM並解析其回應以獲得下一個節點的名稱（例如，"elicit_info"、"recommend_product"）。
5. 返回此名稱，LangGraph將用它來路由執行。

```python
# （LLM和Prompt模板定義會在這裡）

def route_action(state: DialogueState) -> str:
    """
    這是'Think'步驟。
    它分析狀態並決定下一個行動。
    """
    # 1. 檢查缺失的必需slots
    required_slots = ["taste_profile", "caffeine_level"]
    missing_slots = [slot for slot in required_slots if slot not in state["filled_slots"]]

    if missing_slots:
        # 如果缺少slots，行動是引出資訊
        return "elicit_information_node"
    else:
        # 如果所有必需slots都已填滿，行動是推薦
        return "recommend_product_node"

# 在更進階版本中，這裡會進行LLM調用
# 以處理更複雜的路由邏輯，比如識別離題問題。
```

為了簡化，這個初始router使用程式化邏輯。更進階的版本會使用LLM調用來提供更細緻的路由，能夠處理用戶中斷或澄清模糊輸入。

功能節點（"Act"步驟）

接下來，我們定義執行實際工作的節點——「Act」步驟。每個節點是接收DialogueState作為輸入的Python函數，執行行動（通常涉及LLM調用），並返回字典來更新狀態。

- **elicit_information_node**：當router確定缺少必需slot時觸發此節點。它為第一個缺失slot生成澄清問題。

```python
def elicit_information_node(state: DialogueState) -> Dict[str, Any]:
    # （LLM和引導Prompt模板定義會在這裡）
    required_slots = ["taste_profile", "caffeine_level"]
    missing_slots = [slot for slot in required_slots if slot not in state["filled_slots"]]
    slot_to_elicit = missing_slots[0]

    # 使用LLM為slot_to_elicit生成友好問題
    #... llm_call_to_generate_question...
    response_text = f"為了找到完美的咖啡，我需要一點更多資訊。您偏好的{slot_to_elicit.replace('_',

    # 使用AI的問題更新聊天歷史
    new_history = state['chat_history'] + [AIMessage(content=response_text)]
    return {"chat_history": new_history}
```

- **recommend_product_node**：當所有必需slots都已填滿時觸發此節點。它篩選產品目錄並使用LLM生成有說服力的建議。

```python
```

```python
def recommend_product_node(state: DialogueState) -> Dict[str, Any]:
    # （LLM和推薦Prompt模板定義會在這裡）
    filled_slots = state["filled_slots"]

    # 根據filled_slots篩選coffee_knowledge_base
    #... filtering_logic...
    potential_recommendations = [] # 範例結果

    # 使用LLM從篩選列表製作有說服力的建議
    #... llm_call_to_generate_recommendation...
    response_text = f"基於您對{filled_slots['taste_profile']}的偏好，我推薦試試我們的{potential_recor

    new_history = state['chat_history'] + [AIMessage(content=response_text)]
    return {"chat_history": new_history, "recommendations": potential_recommendations}
```

- **handle_user_input_node**：這是處理用戶回應的特殊節點。它嘗試從用戶訊息提取slot資訊並更新狀態中的filled_slots。

```python
def handle_user_input_node(state: DialogueState) -> Dict[str, Any]:
    # （LLM和Slot提取Prompt模板定義會在這裡）
    last_user_message = state['chat_history'][-1].content

    # 使用LLM調用從用戶訊息提取任何slot值
    #... llm_call_to_extract_slots...
    # 範例 extracted_slots = {"taste_profile": "fruity"}
    extracted_slots = {} # 佔位符

    # 更新狀態的filled_slots
    updated_slots = state['filled_slots'].copy()
    updated_slots.update(extracted_slots)
    return {"filled_slots": updated_slots}
```

使用條件邊緣組裝圖

最後，我們將節點添加到圖中並定義連接它們的邊緣。關鍵是從route_action決定點發出的條件邊緣。

```python
# 將節點添加到圖
workflow.add_node("handle_user_input_node", handle_user_input_node)
workflow.add_node("elicit_information_node", elicit_information_node)
workflow.add_node("recommend_product_node", recommend_product_node)

# 入口點是處理用戶輸入以提取slots
workflow.set_entry_point("handle_user_input_node")

# 處理輸入後，我們需要決定接下來做什麼
workflow.add_conditional_edges(
    "handle_user_input_node",
    route_action,
    {
        "elicit_information_node": "elicit_information_node",
        "recommend_product_node": "recommend_product_node"
    }
)

# 引出或推薦後，對話結束這一輪，
# 等待下一個用戶輸入。
workflow.add_edge('elicit_information_node', END)
workflow.add_edge('recommend_product_node', END)

# 將圖編譯為可執行應用
app = workflow.compile()
```

這個圖結構完美實現我們的框架。用戶輸入觸發handle_user_input_node，它更新slots。然後，route_action函數充當「Think」步驟，決定下一步行動。基於其決定，圖轉換到適當的「Act」節點（elicit或recommend），它生成回應並結束轉換。循環準備好用下一個用戶輸入重新開始。

2.4節：實現動態Prompt模板

「Think, Then Act」循環的有效性依賴於在每個階段使用的prompt模板的品質和動態性。這些模板不是靜態的；它們動態地用DialogueState的資訊填充，確保每個LLM調用都精確適應對話的即時語境。這是Dynamic Prompt Adaptation的實際應用。[16]

Router Prompt（"Think"步驟 - 基於LLM的版本）

對於更先進的系統，程式化router可以被基於LLM的router取代。這允許更細緻的決策，如識別離題問題或模糊的用戶陳述。此router的prompt必須受到高度約束以確保它只輸出有效的行動名稱。

```python
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.pydantic_v1 import BaseModel, Field

# 為router定義結構化輸出
class Route(BaseModel):
    """選擇要採取的下一個行動。"""
    action: str = Field(description="要採取的下一個行動。必須是[elicit_information, recommend_prod

# Router Prompt模板
router_prompt_template = ChatPromptTemplate.from_messages([
    ("system", """您是咖啡銷售對話的策略決策者。
    當前已填充的slots：{filled_slots}
    必需的slots：taste_profile, caffeine_level

    分析用戶輸入並選擇下一個行動。"""),
    ("human", "用戶訊息：{user_input}\n\n下一個行動是什麼？")
])

# 此prompt會與支援結構化輸出（tool calling）的LLM一起使用
# 以強制模型輸出有效的'Route'物件。
# llm_with_tools = llm.with_structured_output(Route)
# router_chain = router_prompt_template | llm_with_tools
```

這個prompt將LLM的任務約束為分類問題，使其輸出可靠且直接可用於圖的路由。它接收當前狀態（filled_slots）和用戶輸入作為動態變數。

引導Prompt（"Act"步驟）

elicit_information_node的prompt是動態內容生成的主要範例。它接收slot_to_elicit（由router確定）和來自我們Slot架構表的資訊（如範例問題）來製作自然的查詢。

```python
# 假設slot_schema是從第1.3節表格載入的字典
# slot_schema = {
#   "taste_profile": {"example_question": "例如，您喜歡fruity、chocolatey，還是更earthy的口味？"},
#   ...
# }

elicitation_prompt_template = ChatPromptTemplate.from_messages([
    ("system", """您是友好的咖啡專家，正在幫助客戶找到完美的咖啡。
    您需要詢問關於{slot_to_elicit}的問題。
    使用這個範例作為指導：{example_question}
    保持對話自然且引人入勝。"""),
    ("human", "對話歷史：{chat_history}")
])

# 在elicit_information_node中：
# slot_to_elicit = "taste_profile"
# example_question = slot_schema[slot_to_elicit]["example_question"]
# elicitation_chain = elicitation_prompt_template | llm
# response = elicitation_chain.invoke({
#     "slot_to_elicit": slot_to_elicit,
#     "example_question": example_question,
#     "chat_history": state['chat_history']
# })
```

在這裡，prompt在執行時動態自訂。slot_to_elicit變數告訴LLM要詢問什麼，而example_question提供如何構造問題的關鍵語境。這使prompt高度具體和專注，導致LLM更好且更一致的輸出。

這個兩步prompting機制，其中「Think」prompt的結構化分析輸出（slot_to_elicit）成為「Act」prompt的動態語境輸入，是引導漏斗框架的引擎。它確保代理的對話既策略合理又自然表達，遵循漏斗的商業邏輯而不聽起來機械或重複。[21]

第三部分：進階考量和生產準備

將對話代理從功能原型移至強健的生產就緒應用需要處理現實世界的複雜性。用戶是不可預測的，系統可能失敗，商業需求會演變。這最後部分探索增強引導漏斗框架的進階策

略，專注於優雅的錯誤處理、使用先進AI技術改善建議品質，以及啟用真正個人化的長期記憶。這些考量對創建不僅智能而且可靠、有效和用戶友好的代理至關重要。

3.1節：優雅錯誤處理和對話修復

高品質對話設計的核心原則是系統必須為出錯做好準備。[22]用戶可能提供模糊輸入、離題、表達沮喪或簡單打錯字。生產級代理必須優雅地處理這些情況，而不是失敗或陷入循環。這涉及實施錯誤處理、對話修復和升級策略。

對話修復：這是修復對話中誤解或失誤的過程。[24]對於我們的代理，常見失敗點是handle_user_input_node當LLM無法從用戶訊息提取必需slot時。與其簡單地重複提出同樣問題，更好的方法是進行修復序列：

- **重新表述**：代理可以嘗試重新表述問題。例如，如果「您喜歡什麼樣的風味？」失敗，它可以嘗試「為了縮小範圍，您會說您偏愛像水果一樣的咖啡，還是像巧克力和堅果一樣的？」
- **提供選項**：如果重新表述失敗，代理可以從開放式問題轉為多選格式，呈現可點擊按鈕或編號列表（例如，「沒問題！這些中哪個聽起來最好？1. 果香&明亮，2. 巧克力&豐富，3. 土質&大膽」）。這為用戶簡化任務並使slot-filling過程更強健。[22]

處理中斷：slot-filling過程不應是不可逃脫的陷阱。用戶必須有改變主意或退出流程的自由。[11]系統應該設計為識別和處理中斷意圖。[11]例如，如果用戶說「實際上，算了」或「就給我看最受歡迎的咖啡」，route_action邏輯應該能夠檢測這個意圖並將對話從嚴格的slot-filling序列轉移開。這是基於LLM的router優於純程式化router的地方，因為它可以解釋用戶中斷的語意意圖。我們圖中的handle_off_topic_node將負責優雅地確認用戶請求，並在適當時嘗試將對話引導回漏斗。

升級給人類代理：會有自動代理無法解決的情況。良好的設計會預期這一點並提供清晰的移交升級路徑。[23]升級觸發器可以定義，如連續三次無法理解用戶輸入或檢測到強烈負面情緒（例如，「我真的很沮喪」）。當滿足此觸發器時，代理應該禮貌地提供連接用戶到人類支援專家，確保用戶問題最終得到解決並防止負面客戶體驗。

3.2節：使用Retrieval-Augmented Generation (RAG)增強建議

recommend_product_node的初始實現依賴知識庫的簡單程式化篩選。雖然功能性，這種方法可以透過納入Retrieval-Augmented Generation (RAG)管線顯著改進。RAG透過將LLM生成的回應建基於從外部知識來源檢索的特定資訊來增強品質和事實性。[18]

recommend_product_node的增強工作流程如下：

1. **語意查詢構建**：不只使用filled_slots進行硬篩選，系統會將用戶偏好合成為豐富的語意查詢。例如，偏好{'taste_profile': ['fruity', 'bright'], 'experience_level': 'beginner'}可以轉換為「適合初學者的輕體、酸味和果香咖啡」查詢。
2. **Vector Store檢索**：這是「檢索」步驟。整個咖啡產品目錄——包括詳細描述、客戶評論、品嚐筆記和沖煮指南——會預處理並儲存在vector資料庫（例如，FAISS、Chroma）中。前一步的語意查詢然後用於從此vector store檢索前k個最相關的咖啡文件。[19]這確保代理考慮語意上接近匹配的產品，而不只是確切關鍵字匹配。
3. **增強生成**：這是「生成」步驟。檢索的咖啡文件直接注入建議prompt的語境中，連同用戶原始偏好。prompt會指示LLM作為專家barista，將檢索資訊合成為引人注目的個人化建議。

prompt可能看起來像這樣： 「您是專業咖啡鑑賞家。客戶有以下偏好：{filled_slots}。基於這些偏好和以下檢索的產品資訊，寫個人化建議。解釋*為什麼*推薦的咖啡是很好的匹配，引用檢索語境的具體細節。檢索語境：{retrieved_documents}」

這個基於RAG的方法提供幾個關鍵優勢：

- **減少幻想**：透過強制LLM基於提供的文本進行建議，它大幅減少模型「幻想」或發明產品細節的風險。[15]
- **更豐富的建議**：代理可以生成更有說服力和資訊性的建議，引用檢索文件的具體品嚐筆記、客戶評論或原產地故事。
- **可擴展性**：系統可以簡單地透過將新產品添加到vector store來更新，而無需重新訓練或修改LLM prompts的核心邏輯。

3.3節：為長期記憶和個人化持久化狀態

真正出色的對話代理應該能夠隨時間與用戶建立關係。雖然LLMs本質上是無狀態的，意味著每個請求都被獨立處理，但我們設計的架構不是。[21]DialogueState物件和LangGraph框架為單一會話期間提供內建有狀態性。[20]要實現真正的長期記憶，此狀態必須在會話間持久化。

策略涉及在每次對話結束時將DialogueState物件保存到外部資料庫（如Redis等鍵值儲存或傳統SQL資料庫）。每個狀態會用唯一用戶識別符（例如，來自登入系統的用戶ID或會話cookie）作為鍵。

當用戶返回應用時，系統首先檢查資料庫中是否有與其ID關聯的持久化狀態。如果找到，它會被載入為新對話的初始狀態。這個簡單機制解鎖個人化和關係建立的強大功能[25]：

- **個人化問候**：代理可以按姓名問候回歸用戶（例如，「歡迎回來，Jane！」）。
- **語境連續性**：代理可以參考過去互動。例如，它可能會問「上次您購買了我們的Ethiopia Yirgacheffe。您今天在找類似的東西，還是想試試新的？」
- **主動建議**：透過分析用戶購買歷史和先前陳述的偏好（全部儲存在其持久化狀態的filled_slots中），代理可以主動提供量身定制的建議，而無需每次都經過整個slot-filling漏斗。
- **建立信任**：記住用戶及其偏好使互動感覺更個人化且較少交易性，培養信任和忠誠度，這是成功對話漏斗的關鍵目標。[1]

像LangGraph Platform這樣的生產級平台提供處理狀態持久化、檢查點和長期記憶的管理解決方案，簡化這些先進功能的部署。[20]透過實現狀態持久化，對話代理從單次使用工具進化為持續學習的伴侶，隨每次互動改善其服務。

結論

多輪引導漏斗框架代表商業對話式AI的重大架構演進。透過超越簡單的反應式問答，這個模型使創建能夠策略性引導用戶達成特定商業目標的主動代理成為可能。這個框架的核心在於透過使用prompt chaining的「Think, Then Act」循環實現的認知功能的深思熟慮分離。這個兩步過程——系統首先分析對話狀態以決定策略行動，然後基於該決定生成面向用戶的回應——提供了執行複雜商業邏輯同時利用現代LLMs細緻語言能力的強健機制。

使用Python、LangChain和LangGraph的實際實現展示了這種方法的可行性和力量。特別是LangGraph，證明是管理目標導向對話的有狀態、循環和條件性質的不可或缺工具。DialogueState的詳細構建、基於router的控制流程和動態prompt模板為尋求構建類似系統的開發者提供清晰藍圖。

此外，錯誤處理、RAG增強建議和持久狀態的先進考量突出了從功能原型到生產就緒應用的路徑。成功的對話代理必須具有彈性、事實根據並能夠隨時間建立個人化關係。透過整合這些功能，代理超越其作為簡單銷售工具的角色，成為智能、可靠且引人入勝的品牌代表。

最終，引導漏斗框架提供結構化且可擴展的方法論，用於將大型語言模型的力量與企業的具體目標結合，為新一代更有效且複雜的對話體驗鋪平道路。