MGFD Design

version 0.2 2025/08/11

Chat-Turn概念:

Simple Chat Turn: A chat between user and LLM with no any objective.

LLM-Self Response Chat Turn

Prompt-Generation Chat Turn

核心模組的啟動與運作順序

系統的運作並非單一的「輸入->輸出」過程，而是遵循一個預先設計好的狀態圖 (State Graph)。針對此查詢，核心模組的啟動順序如下：

1. **`UserInputHandler`** **(使用者輸入處理模組)**：系統的入口點。首先接收並解析使用者的原始文字輸入。
2. **`DialogueManager (Router)`** **(對話管理器 - 路由節點)**：系統的「大腦」或「思考」環節。分析第一步處理後的結果，決定下一步應採取的策略。
3. **`ActionExecutor (Elicitation Node)`** **(動作執行器 - 資訊引導節點)**：系統的「執行」環節。根據大腦的決策，執行具體動作，在此案例中為「向使用者提問以獲取更多資訊」。
4. **`ResponseGenerator`** **(回應生成模組)**：系統的「口舌」。將執行結果包裝成前端介面可以理解的格式並輸出。

整個流程構成一個完整的對話回合 (turn)，系統在輸出回應後，會等待使用者的下一次輸入，並重複此循環。

核心模組的啟動與運作順序系統的運作並非單一的「輸入->輸出」過程，而是遵循一個預先設計好的狀態圖 (State Graph)。針對此查詢，核心模組的啟動順序如下：

a. UserInputHander With Slot-Filling: (使用者輸入處理模組)：系統的入口點。首先接收並解析使用者的原始文字輸入。

✳ 溝通與接收：

- 溝通函式：`on_user_submit(raw_text: str)`
- 接收資料：函式接收到由前端介面傳來的使用者原始查詢字串："請幫我介紹適合業務工作，快速開關機，且攜帶方便的筆電"。

✳ 模組內部運算：

1. **`extract_slots_from_text(text)`** 函式：此函式是本模組的核心。它會調用一個大型語言模型 (LLM)，並給予一個包含「筆電銷售槽位綱要 (Slot Schema)」的提示 (Prompt)。
2. 槽位綱要 **(Slot Schema)** 定義：系統預先定義了推薦筆電所需的多個資訊槽位，例如：`use_case` (使用情境), `portability` (便攜性), `performance_features` (效能特點), `budget` (預算), `screen_size` (螢幕尺寸) 等。
3. **LLM** 進行資訊萃取：LLM 根據提示，從使用者輸入的文字中，將對應的資訊填入相應的槽位。

✳ 轉換與傳遞：

- 轉換後資料結構：運算完成後，原始文字被轉換成一個結構化的 Python `dict` (字典) 或 JSON 物件，代表目前已填滿的槽位。

```
{
"use_case": "業務工作",
"portability": "高",
"performance_features": ["快速開關機"]
```

```
}
```

✳ 傳遞給下一模組:

此字典會用於更新系統全域的 `DialogueState` (對話狀態) 物件。整個更新後的 `DialogueState` 物件被傳遞給下一個核心模組 `DialogueManager`。

---

b. Dialogue Manager(Think+Router): 對話管理器 - 路由節點

✳ 溝通與接收:

- 溝通函式: `route_next_action(state: DialogueState)`
- 接收資料: 接收包含前一步驟更新結果的完整 `DialogueState` 物件。

✳ 模組內部運算:

1. **`check_required_slots(state)` 函式**: 此函式是本模組的決策核心。它會檢查 `DialogueState` 中的 `filled_slots` (已填槽位), 並與預先定義的「必要槽位 (Required Slots)」列表進行比對。
2. 決策邏輯: 系統的「筆電銷售漏斗」規則定義了 `budget` (預算) 和 `screen_size` (螢幕尺寸) 為完成推薦的必要槽位。函式發現這兩個槽位在目前的 `DialogueState` 中是空的。
3. 產生決策: 基於上述發現, 函式判定對話漏斗無法繼續向下, 必須先獲取缺失的必要資訊。因此, 它做出的決策是「引導使用者提供資訊 (Elicit Information)」。

✳ 轉換與傳遞:

- 轉換後資料結構: 運算結果是一個指令型別的字串或物件, 清晰地指示了下一步的行動。

```
{
  "action": "ELICIT_SLOT",
  "parameter": "budget"
}
```

(指令:引導使用者提供資訊, 目標槽位:預算)

- 傳遞給下一模組: 此指令物件被傳遞給 `ActionExecutor` 模組, 由它來執行具體動作。

---

c. ActionExecutor (Elicitation Node): 動作執行器 - 資訊引導節點。

✳ 溝通與接收:

- 溝通函式: `execute_action(command: dict, state: DialogueState)`
- 接收資料: 接收來自 `DialogueManager` 的指令物件 `{"action": "ELICIT_SLOT", "parameter": "budget"}`, 並同時存取目前的 `DialogueState` 以獲得對話上下文。

✳ 模組內部運算:

1. `generate_elicitation_prompt(parameter, chat_history)` 函式：根據收到的指令，此函式會動態生成一個新的提示給 LLM。
2. 動態提示內容：提示會指示 LLM：「你是一位專業的電腦銷售員。根據目前的對話歷史，用親切自然的語氣，向使用者詢問關於他的『預算』資訊。」
3. **LLM** 生成回應文本：LLM 根據此高度聚焦的任務，生成一段自然的問句。

轉換與傳遞：

- 轉換後資料結構：LLM 生成的純文字會被包裝成一個更豐富的回應物件，其中可能包含給前端的建議選項，以優化使用者體驗。

```
{
"response_type": "text_with_suggestions",
"content": "好的，了解您需要一台適合業務工作、輕便且反應快速的筆電。為了給您更精準的推薦，請問您的預算大概是多少呢？",
"suggestions": ["約 3-4 萬", "約 4-5 萬", "5 萬以上"]
}
```

✳ 傳遞給下一模組：

此回應物件被傳遞給最後的 `ResponseGenerator` 模組進行輸出。

最終回應生成與前端溝通

- 負責模組：`ResponseGenerator` (回應生成模組)

- 處理的資料結構：此模組接收到由 `ActionExecutor` 傳來的、結構化的回應物件。這個物件（如上所示）已經包含了所有前端介面需要渲染的資訊，包括主要回覆文本和可點擊的建議按鈕。
- 最終輸出格式與前端溝通：
    1. 序列化 **(Serialization)**：`ResponseGenerator` 模組的主要職責是將內部的 Python 回應物件序列化為 **JSON** 格式。這是目前 Web 應用中最通用的前後端數據交換格式。
    2. **API** 端點：系統後端會提供一個 API 端點 (例如 `/api/chat`)。當前端發送使用者查詢後，會持續等待此端點的回應。
    3. 溝通方式：`ResponseGenerator` 將序列化後的 JSON 字串透過 HTTP 回應 (HTTP Response) 的主體 (Body) 回傳給前端。
    4. 前端渲染：前端應用程式接收到這個 JSON 後，會解析其內容：
        - 讀取 `content` 欄位，將其渲染成一個 AI 助手的對話氣泡。
        - 讀取 `suggestions` 陣列，並動態生成三個可點擊的按鈕。

透過這種方式，系統不僅回答了使用者的問題，更以結構化、可控的方式推動對話，朝著完成「成功推薦筆電」這個最終業務目標邁進。

展示範例：

依照先前報告中的架構，使用 Python 實作一個完整的、可運行的範例。在這個範例中，我們將使用一個 JSON 檔案 (`session_db.json`) 來模擬 Redis 或其他 Key-Value 資料庫，用以儲存和讀取對話狀態，完整地展示從接收使用者輸入到最終生成回應的整個流程。

```
import json
import os
from typing import TypedDict, List, Dict, Any, Literal
```

```
# --- 1. 系統設定與資料結構定義 ---

# 模擬前端傳來的原始使用者輸入
USER_QUERY = "請幫我介紹適合業務工作，快速開關機，且攜帶方便的筆電"
SESSION_ID = "user_abc_123"  # 每個使用者一個獨立的 Session ID
DB_FILE = "session_db.json" # 用於模擬 Redis 的 JSON 資料庫檔案

# 定義「對話狀態」的資料結構，與報告中一致
class DialogueState(TypedDict):
    """
    代表一個獨立 session 的完整對話狀態。
    """
    session_id: str
    chat_history: List]
    filled_slots: Dict[str, Any]

# 定義筆電銷售所需的「槽位綱要 (Slot Schema)」
# 'required' 標記了是否為完成推薦的必要資訊
SLOT_SCHEMA = {
    "use_case": {"type": "String", "required": True},
    "portability": {"type": "String", "required": True},
    "performance_features": {"type": "List", "required":
False},
    "budget": {"type": "String", "required": True},
    "screen_size": {"type": "String", "required": True},
}
```

```
# --- 2. 模擬資料庫 (JSON as Redis) ---

def load_session_state(session_id: str) -> DialogueState:
    """
    從 JSON 檔案中讀取指定 session_id 的對話狀態。
    如果檔案或 session 不存在，則回傳一個全新的初始狀態。
    """
    if not os.path.exists(DB_FILE):
        return DialogueState(session_id=session_id,
chat_history=, filled_slots={})

    with open(DB_FILE, 'r', encoding='utf-8') as f:
        try:
            data = json.load(f)
            state_data = data.get(session_id)
            if state_data:
                return DialogueState(**state_data)
        except json.JSONDecodeError:
            pass # 檔案為空或格式錯誤，將回傳新狀態

    # 如果找不到對應的 session，回傳一個新的狀態
    return DialogueState(session_id=session_id, chat_history=,
filled_slots={})

def save_session_state(state: DialogueState):
    """
```

```
    將更新後的對話狀態寫回 JSON 檔案。
    """
    all_data = {}
    if os.path.exists(DB_FILE):
        with open(DB_FILE, 'r', encoding='utf-8') as f:
            try:
                all_data = json.load(f)
            except json.JSONDecodeError:
                all_data = {} # 檔案損毀或為空

    all_data[state['session_id']] = state

    with open(DB_FILE, 'w', encoding='utf-8') as f:
        json.dump(all_data, f, ensure_ascii=False, indent=4)

# --- 3. 模擬大型語言模型 (LLM) 的函式 ---
# 在真實世界中，這些函式會是 API 呼叫

def mock_llm_extract_slots(text: str) -> Dict[str, Any]:
    """
    模擬 LLM 接收原始文本，並根據 Slot Schema 萃取出資訊。
    """
    print("\n 正在萃取使用者輸入中的槽位...")
    # 根據使用者輸入，LLM 辨識出以下資訊
    extracted = {
      "use_case": "業務工作",
      "portability": "高", # "攜帶方便" 被 LLM 解讀為 "高"
```

```
        "performance_features": ["快速開關機"]
    }
    print(f" 萃取結果: {extracted}")
    return extracted

def mock_llm_generate_elicitation_response(slot_to_elicit: str,
chat_history: List) -> Dict[str, Any]:
    """
    模擬 LLM 根據「需要詢問的槽位」和「對話歷史」，生成一個自然的回應。
    """
    print(f"\n 正在生成關於 '{slot_to_elicit}' 的提問...")
    # 這裡可以加入更複雜的邏輯，根據不同 slot 生成不同問題
    if slot_to_elicit == "budget":
        response = {
          "response_type": "text_with_suggestions",
          "content": "好的，了解您需要一台適合業務工作、輕便且反應快
速的筆電。為了給您更精準的推薦，請問您的預算大概是多少呢？",
          "suggestions": ["約 3-4 萬", "約 4-5 萬", "5 萬以上"]
        }
        print(f" 生成的回應物件: {response}")
        return response
    # 可以為其他 slot 增加更多情境
    return {"content": f"請問您的 {slot_to_elicit} 是？",
"suggestions":}


# --- 4. 核心模組實作 ---
```

```python
def user_input_handler(raw_text: str, current_state:
DialogueState) -> DialogueState:
    """
    模組一:使用者輸入處理模組
    接收原始文字,調用 LLM 萃取槽位,並更新對話狀態。
    """
    print("\n--- 模組 1: UserInputHandler 啟動 ---")

    # 1. 接收資料:raw_text
    print(f"接收到原始輸入: '{raw_text}'")

    # 2. 內部運算:調用 LLM 進行資訊萃取
    extracted_slots = mock_llm_extract_slots(raw_text)

    # 3. 轉換與傳遞:更新 DialogueState
    updated_state = current_state.copy()
    updated_state["chat_history"].append({"role": "user",
"content": raw_text})
    updated_state["filled_slots"].update(extracted_slots)

    print(f"狀態更新完成,準備傳遞給下一模組。")
    return updated_state


def dialogue_manager_router(current_state: DialogueState) ->
Dict[str, str]:
```

```
    """
    模組二:對話管理器 (路由節點)
    系統的「大腦」,分析當前狀態,決定下一步行動。
    """
    print("\n--- 模組 2: DialogueManager (Router) 啟動 ---")
    print("正在分析 DialogueState 以決定下一步行動...")

    # 1. 接收資料:current_state
    filled_slots = current_state["filled_slots"]

    # 2. 內部運算:檢查是否有「必要」槽位尚未被填寫
    for slot_name, schema in SLOT_SCHEMA.items():
        if schema["required"] and slot_name not in
filled_slots:
            print(f"決策:發現必要槽位 '{slot_name}' 缺失。")
            # 3. 轉換與傳遞:回傳指令
            decision = {"action": "ELICIT_SLOT", "parameter":
slot_name}
            print(f"生成指令: {decision}")
            return decision

    # 如果所有必要槽位都已填寫
    print("決策:所有必要槽位皆已滿足。")
    decision = {"action": "RECOMMEND_PRODUCT", "parameter":
None}
    print(f"生成指令: {decision}")
    return decision
```

```
def action_executor(command: Dict[str, str], current_state:
DialogueState) -> (Dict, DialogueState):
    """
    模組三：動作執行器
    根據大腦的指令，執行具體動作。
    """
    print("\n--- 模組 3: ActionExecutor 啟動 ---")
    print(f"接收到指令: {command}")

    action = command.get("action")
    parameter = command.get("parameter")

    if action == "ELICIT_SLOT":
        # 1. 接收資料：command, current_state
        # 2. 內部運算：調用 LLM 生成提問
        response_object =
mock_llm_generate_elicitation_response(parameter,
current_state["chat_history"])

        # 3. 轉換與傳遞：更新對話歷史並準備最終輸出
        updated_state = current_state.copy()
        updated_state["chat_history"].append({"role":
"assistant", "content": response_object["content"]})

        return response_object, updated_state
```

```
    elif action == "RECOMMEND_PRODUCT":
        # 在此處實作推薦邏輯
        response_object = {"content": "根據您的需求，我推薦以下產品...", "suggestions":}
        updated_state = current_state.copy()
        updated_state["chat_history"].append({"role": "assistant", "content": response_object["content"]})
        return response_object, updated_state

    # 處理未知指令
    return {"content": "抱歉，我現在有點問題，請稍後再試。"}, current_state


def response_generator(response_object: Dict) -> str:
    """
    模組四：回應生成模組
    將內部回應物件序列化為 JSON，準備傳給前端。
    """
    print("\n--- 模組 4: ResponseGenerator 啟動 ---")
    print("正在將最終回應物件序列化為 JSON...")

    # 序列化為格式化的 JSON 字串
    json_output = json.dumps(response_object, ensure_ascii=False, indent=4)
```

```
    print("序列化完成。")
    return json_output

# --- 5. 主執行流程 ---

def main():
    """
    主函式，模擬一個完整的使用者互動回合。
    """
    print("="*50)
    print("AI-Sale System 啟動一個新的對話回合")
    print(f"Session ID: {SESSION_ID}")
    print("="*50)

    # 1. 從資料庫載入此 session 的狀態
    state = load_session_state(SESSION_ID)

    # 2. 模組一：處理使用者輸入
    state = user_input_handler(USER_QUERY, state)

    # 3. 模組二：決定下一步行動
    command = dialogue_manager_router(state)

    # 4. 模組三：執行指令
    final_response_object, state = action_executor(command,
state)
```

```
    # 5. 模組四：生成給前端的最終 JSON
    json_to_frontend =
response_generator(final_response_object)

    # 6. 將更新後的狀態存回資料庫
    save_session_state(state)
    print(f"\n 已將 Session '{SESSION_ID}' 的最新狀態存回
'{DB_FILE}'")

    print("\n" + "="*50)
    print("流程結束。以下是最終產出：")
    print("="*50)

    print("\n【產出 1: 準備傳送給前端的 JSON】")
    print(json_to_frontend)

    print("\n【產出 2: 儲存在資料庫中的最新對話狀態】")
    with open(DB_FILE, 'r', encoding='utf-8') as f:
        print(f.read())

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

---

d. (optional not implement now) Long Term Memory

e. (optional not implement now)
Recommendation System Concepts Integration

結後語:

這個引導式漏斗框架(Multiturn Guided Funnel Framework)透過從根本上改變對話式 AI 的設計理念，從而被動回答問題轉變為主動引導用戶達成特定商業目標，例如銷售和客戶服務中的有價值的成果。

以下是此框架如何轉變現有對話式 AI 以實現商業目標的幾個關鍵方面:

- 從「拉取」模型轉變為「推送」模型 **(Shift from "Pull" to "Push" Model)**:
    - 傳統的 Retrieval-Augmented Generation (RAG) 應用通常採用「拉取」模型，即等待用戶查詢並檢索相關答案，用戶完全控制對話方向。
    - 而引導式漏斗框架則採用「推送」模型，代理會主動將對話導向預定目標，例如「引導潛在客戶」、「發起互動」和「培育潛在客戶」。這要求更複雜的架構來維持目標、制定策略並根據用戶反應進行調整。
- 複製傳統銷售漏斗流程 **(Replicating the Sales Funnel)**:
    - 「對話漏斗」將傳統的行銷和銷售漏斗階段(如認知、興趣、評估、參與、行動和保留)直接轉換到自動化對話領域。
    - 對話代理不再是被動的資訊儲存庫，而是主動的策略指南，系統性地引導用戶經歷這些階段。例如，透過歡迎用戶建立認知，提出問題激發興趣，提供資訊進行評估，提供促銷進行參與，直接在聊天中促成行動，並在購買後提供支援以促進保留。
- 實現顯著商業優勢 **(Yielding Significant Business Advantages)**:

    - 實施此漏斗模型帶來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等顯著優勢。
    - 透過自動化資格審查和培育流程，銷售團隊可以專注於高價值活動，從而加快交易完成並大幅節省成本。
- 核心架構元件支持目標導向 **(Core Architectural Components for Goal-Oriented Design)**:
    - 對話管理器 **(Dialogue Manager, DM)**：作為系統的「認知核心」，它管理對話狀態並控制對話流程。DM 包含對話狀態追蹤 **(Dialogue State Tracking, DST)**，用於維護對話的結構化記憶，以及對話控制 **(Dialogue Control)**，用於基於當前狀態決定下一步行動，例如提問、推薦或升級給人類代理。
    - 「思考，然後行動」循環 **("Think, Then Act" Cycle)**：這是該框架的核心實現，將 DM 的功能分解為兩個由 LLM 驅動的步驟，以確保可靠性和可控性。
        - 「思考」階段 **(Think Phase)** (prompt-lvl1)：LLM 作為推理引擎，分析對話狀態並決定下一步的策略性行動，其輸出是結構化指令（如 JSON 物件），而非用戶可見的文本。
        - 「行動」階段 **(Act Phase)** (prompt-lvl2)：另一個 LLM 調用將「思考」階段的結構化指令轉化為自然、引人入勝且面向用戶的回應。這種分離限制了 LLM 在每個步驟的任務，減少了其「幻想」或偏離商業邏輯的風險。
    - 槽位填充 **(Slot Filling)**：這是推動對話漏斗前進的主要機械過程，用於收集完成任務所需的特定資訊。每個填滿的槽位都代表用戶深入漏斗一步，將通用查詢轉化為具體的偏好。系統會主動識別並生成針對性問題來引導用戶填寫缺失的必需槽位。
- 高級考量與生產準備 **(Advanced Considerations for Production Readiness)**:

- 優雅的錯誤處理和對話修復：系統設計用於處理模糊輸入、離題、用戶挫敗或打字錯誤。透過重新表述問題、提供多選選項或在無法解決時升級給人類代理，確保了流暢的用戶體驗並防止負面客戶體驗。
- 使用 **RAG** 增強建議：透過將用戶偏好合成為語義查詢，從向量資料庫中檢索最相關的產品文檔，並將其注入到推薦提示中，LLM 可以生成更豐富、更具說服力且事實更準確的個性化推薦，同時減少幻覺的風險。
- 持久化狀態實現長期記憶和個性化 **(Persisting State for Long-Term Memory and Personalization)**：儘管 LLM 本質上是無狀態的，但透過將 `DialogueState` 對象保存到外部資料庫並在用戶返回時載入，代理可以實現長期記憶。這使得個性化問候、語境連續性、主動推薦和建立用戶信任與忠誠度成為可能，這是一個成功的對話漏斗的關鍵目標。

儘管任何複雜系統的實作都會面臨挑戰，例如需要仔細的對話設計、精確的 Slot Schema 定義、高品質的知識庫 以及熟練的開發團隊來利用 LangChain 和 LangGraph 的功能，但框架預先考慮並提供了解決這些問題的策略。因此，如果能夠投入足夠的資源和專業知識進行嚴謹的規劃與實作，多輪引導漏斗框架實作成功的可能性非常高，並且能夠有效且策略性地幫助企業達成其商業目標。