MODULE 1 UNIT 6

LangChain Practice

Ver. 1.0

Table of contents

1. LangChain Practice

1.1 Advanced Customer Service Agent with LangChain

Evolve the "Automated Customer Service System" from the initial Prompt Engineering practice by rebuilding it as a robust, multi-step agent using the **LangChain** framework. This challenge will focus on creating a composable, observable, and reliable application by leveraging LangChain Expression Language (LCEL), Pydantic for structured data handling, and LangSmith for tracing and debugging.

1.2 Fictional Company Context

You will continue to work with the context of **TechStore Plus**. All company information, product catalogs, and policies remain the same as in the "Prompt Engineering Practice" module.

- **Company:** TechStore Plus - Your Trusted Technology Store
- **Sector:** E-commerce for technology products
- **Location:** New York, USA
- **Key Services:** Sales, Technical Support, Warranty, Financing, Trade-ins.
- **Key Policies:** Shipping, Returns, Installation, Extended Warranty.

1.3 System Architecture & Components

You will develop a single, cohesive LangChain chain that processes a user's query through multiple stages. The use of **LangChain Expression Language (LCEL)** is required to pipe the components together.

1.3.1 Component 1: Query Analysis & Classification

This component's objective is to analyze the initial user query and extract key information into a structured format.

Requirements:

- Create a Pydantic **BaseModel** to define the schema for the analysis.
- Use a LangChain Chat Model with the **.with_structured_output()** method to force the output into your Pydantic object.
- The Pydantic model must include the following fields:
    - **query_category:** (Literal) Classify the query into one of: "technical_support", "billing", "returns", "product_inquiry", "general_information".
    - **urgency_level:** (Literal) "low", "medium", "high".
    - **customer_sentiment:** (Literal) "positive", "neutral", "negative".
    - **entities:** A Pydantic sub-model or dictionary to extract key entities like product_name, order_number, date, if present.

Example Pydantic Schema:

```
from pydantic import BaseModel, Field
from typing import Literal, Optional

class ExtractedEntities(BaseModel):
    product_name: Optional[str] = Field(None, description="The specific
product mentioned by the user")
    order_number: Optional[str] = Field(None, description="The order
number mentioned by the user")

class QueryAnalysis(BaseModel):
    """Analyzes and classifies a customer query."""
    query_category: Literal["technical_support", "billing", "returns",
"product_inquiry", "general_information"]
    urgency_level: Literal["low", "medium", "high"]
    customer_sentiment: Literal["positive", "neutral", "negative"]
    entities: ExtractedEntities
```

1.3.2 Component 2: Dynamic Response Generation

This component will take the structured output from Component 1 and generate a context-aware, personalized response.

Requirements:

- Implement logic (e.g., using a RunnableLambda or a custom function within your chain) to dynamically select the next step based on the query_category.
- Create different PromptTemplate instances for different categories to guide the response generation.
    - For example, a "technical_support" prompt should be empathetic and ask for troubleshooting steps, while a "product_inquiry" prompt should be helpful and informative.
- The response generation prompt must receive the original user query AND the analysis from Component 1.
- The generated response must:
    - Match the customer's sentiment (e.g., be more apologetic for a negative sentiment).
    - Acknowledge extracted entities (e.g., "I'm sorry to hear about the issue with your order #TEC-2024-001...").
    - Provide clear, actionable next steps for the user.

1.3.3 Component 3: Conversation Summarization & Persistence

The final step of your chain will be to generate a structured summary of the entire interaction, ready for logging.

Requirements:

- Create a final Pydantic BaseModel that matches the JSON structure specified in the original "Prompt Engineering Practice" document.
- The final chain should output an instance of this Pydantic model. The model should be populated using information gathered throughout the chain (the initial analysis, the final response, etc.).

Required Pydantic Structure:

```
from pydantic import BaseModel
from typing import List

class ConversationSummary(BaseModel):
    """A structured summary of the customer service interaction."""
    timestamp: str
    customer_id: str = "auto_generated"
    conversation_summary: str = Field(description="A concise,
one-sentence summary of the interaction.")
    query_category: str
    customer_sentiment: str
    urgency_level: str
    mentioned_products: List[str]
    extracted_information: dict
    resolution_status: Literal["resolved", "pending", "escalated"]
    actions_taken: List[str] = Field(description="A list of actions the
agent took or suggested.")
    follow_up_required: bool
```

1.4 LangSmith Integration

Observability is crucial. You must use LangSmith to monitor and debug your application.

Requirements:

- Configure the necessary environment variables to enable tracing for your project.
- Set a clear project name in LangSmith (e.g., "Advanced-Customer-Agent").
- After running your test queries, inspect the traces in the LangSmith UI. Pay close attention to the multi-step nature of your chain, observing the inputs and outputs of each component.
- **Bonus Challenge:** Create a small evaluation dataset in LangSmith with 5-10 examples from the test queries. Run your chain over this dataset and add a simple "Correctness" feedback score to each run to evaluate if your agent's query_category classification was accurate.

1.5 Deliverables

The complete practical work will be developed in a single Jupyter Notebook containing:

- Clearly identified sections with markdown for each architectural component.

- Implementation using LangChain Expression Language (LCEL) to build the agent.
- Pydantic models for QueryAnalysis and ConversationSummary fully integrated into the chain.
- Usage examples for each suggested test query, demonstrating the end-to-end functionality.
- Explanatory comments in each code cell, explaining the purpose of different parts of the chain and the prompting techniques used.
- Results analysis using markdown between code cells. Include a screenshot or a public link to a LangSmith trace for one of the complex queries (e.g., "Urgent-Negative") to demonstrate successful tracing.

1.6 Suggested Test Queries:

Use the same sample data from the previous practice module to test your new LangChain-powered agent.

1. Neutral-Informative: "Hello, I'd like to know if you have the new iPhone 15 in stock and how much shipping costs to Chicago"
2. Urgent-Negative: "This is an emergency! My order #TEC-2024-001 never arrived and I need that laptop for work tomorrow!"
3. Satisfied-Positive: "Thank you so much for the excellent service with my previous purchase, I want to buy gaming headphones"
4. Frustrated-Technical: "I can't configure the router I bought last week, I've tried everything and it doesn't work"
5. Formal-Billing: "Good morning, I need the receipt for my purchase from December 15th, order #TEC-2023-089"
6. Warranty-Query: "I bought a tablet 8 months ago and now it won't turn on, how do I use the warranty?"