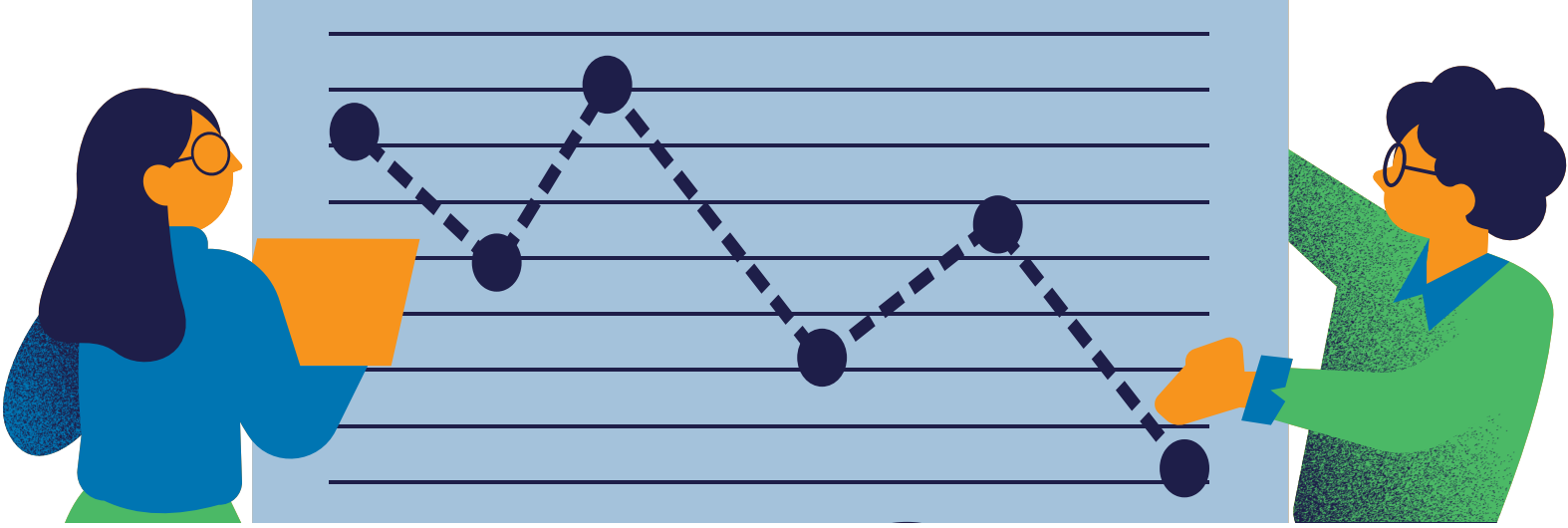


Data Analytics AI Agent using LangChain

Created by: Sowmily Dutta

1. Problem Statement

Objective:

The **Data Analytics AI Agent** project addresses the critical challenge of making structured data analysis accessible to users without technical coding skills. It provides a conversational interface to a CSV dataset (Netflix catalog data), allowing users to pose complex questions in natural language. The system is designed to interpret these queries and deliver two distinct types of outputs:

1. Factual Insights: Direct textual answers derived from data aggregation, filtering, or calculations (e.g., "How many movies were released in 2020?").
2. Data Visualizations: Generated charts and graphs based on user requests (e.g., "Plot a graph of content ratings."), saved as image files.

The project's core objective is to abstract away the complexity of pandas and Matplotlib code, empowering users to perform data analytics through simple conversation.

LangChain Justification:

LangChain is an ideal and exceptionally well-chosen framework for this problem, and its use is thoroughly justified by the project's architecture:

- High-Level Agentic Abstractions: The project's foundation is the create_pandas_dataframe_agent. This is a powerful, high-level LangChain abstraction that encapsulates a complex "Reason-Act" (ReAct) workflow. It saves the developer from manually implementing the entire loop of interpreting a query, generating Python code, executing it in a safe environment (a Python REPL), and parsing the output. LangChain provides this sophisticated capability out-of-the-box.
- Component Ecosystem & Modularity: The project's strength lies in its modular, multi-agent design. LangChain's component-based nature makes this elegant architecture possible. The system effectively utilizes:
  - Agents & Toolkits: The langchain_experimental.agents.agent_toolkits provides the specialized pandas agent, which is the perfect tool for interacting with a DataFrame.

    - LLM Wrappers: The langchain_openai.OpenAI wrapper provides a standardized, simple interface to the language model, handling API calls and configuration seamlessly.
    - Prompt Engineering: The langchain_core.prompts.PromptTemplate is used to create structured, reusable, and highly specific instructions for the agents, separating the core logic of the prompt from the application code. This is most evident in the charting_agent and conversational_agent.
- LangChain Expression Language (LCEL): The conversational_agent is constructed using LCEL (prompt | llm). This demonstrates an understanding of modern LangChain best practices, creating a transparent, composable, and efficient chain for the formatting task, which is superior to legacy chain implementations.

2. Overall System Design

Innovation & Creativity:

The project's innovation lies not in the use of a single pandas agent, but in its creative multi-agent orchestration with specialized roles and intelligent output processing.

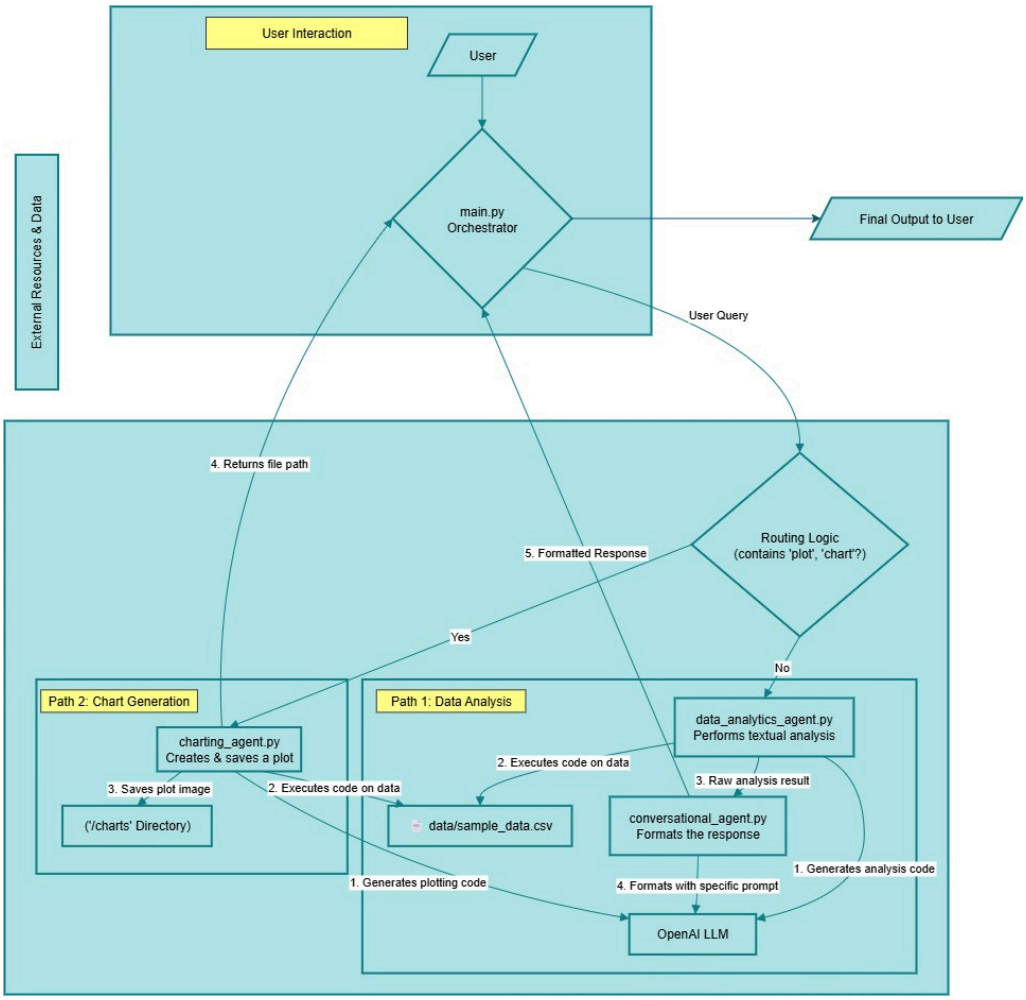
- Agent Specialization: The design cleverly avoids a "one-size-fits-all" approach. By creating a data_analytics_agent for general queries and a separate charting_agent with a highly tailored prompt, the system achieves greater reliability and control for distinct tasks. This separation of concerns is an elegant architectural pattern.
- Pragmatic Routing: The use of a simple, keyword-based router in main.py is a pragmatic and efficient design choice. It avoids the latency and cost overhead of using another LLM call for routing, which is perfectly suitable for the defined problem scope.
- Output Enhancement Agent: The most creative element is the inclusion of the conversational_agent as a post-processing step. The raw output from the pandas agent can often be unpolished (e.g., just a number, or a Python list representation). By piping this raw result through a final formatting agent, the system guarantees a consistently user-friendly and well-presented response, dramatically improving the user experience.

Optimum Architecture, Workflow, and Implementation:

The overall architecture is well-structured, logical, and highly effective for the problem at hand.

- Architecture: The project employs a clean, layered architecture:
    - Orchestration/Interface Layer (main.py): Manages the user interaction loop and routing logic.
    - Agent Logic Layer (agents/ directory): Encapsulates the core intelligence, with each agent's creation and querying logic separated into its own module.
    - Configuration Layer (config.py): Centralizes constants like file paths and keywords, which is excellent for maintainability.
- Workflow: The system follows a clear and logical data flow:
    1. The user enters a query.
    2. The main function acts as a router, checking the query for charting keywords.
    3. Charting Path: If a keyword is found, the query is sent to the charting_agent, which is specifically prompted to generate Python code, save a plot to the /charts directory, and return the file path.
    4. Data Analysis Path: If no charting keyword is found, the query is sent to the data_analytics_agent. This agent generates and executes pandas code to find the answer.
    5. The raw output from the data agent is then passed to the conversational_agent.
    6. This final agent reformats the raw data into a polished, natural-language sentence.
    7. The final, clean response is displayed to the user.

Flowchart:



Appropriate Use of LangChain Components:

The project demonstrates a mature understanding of selecting the right LangChain component for the right task.

- create_pandas_dataframe_agent: This is the ideal component for this project, as the core task is to interact with a pandas DataFrame. It provides the necessary tools and reasoning capabilities.
- OpenAI (LLM): The chosen LLM wrapper. The decision to set temperature=0 is a crucial and correct choice for data analysis, ensuring the agent prioritizes factual accuracy and deterministic outputs over creativity.
- PromptTemplate: Used effectively in the conversational_agent and implicitly via a formatted string in the charting_agent. This separates prompt logic from execution code, making the system easier to debug and maintain.
- RunnableSequence (LCEL): The use of the | pipe operator to build the conversational_agent showcases adherence to modern LangChain best practices, resulting in a more declarative and transparent chain.

3. Code Implementation

Well-documented End-to-End Code:

The code is exceptionally well-documented.

- Each function has a clear docstring explaining its purpose, arguments, and return values, following best practices.
- Inline comments are used effectively to explain important decisions, such as We are using a temperature of 0 to get deterministic outputs.
- The prompt within query_charting_agent is a prime example of excellent documentation, providing clear, numbered instructions that guide the LLM's behavior and make the agent's logic transparent.

Appropriate Function and Layer Design:

The code structure is exemplary.

- Separation of Concerns: A clear separation exists between the main application loop (main.py), configuration (config.py), and agent-specific logic (agents/). This modularity makes the project easy to understand, test, and extend.
- Function Granularity: Functions are well-defined and have a single responsibility. For example, create_..._agent functions are responsible only for instantiation, while query_..._agent functions handle the invocation. This is a clean design.

Code Quality & Readability:

- Modularity: The code is highly modular. One could easily replace the OpenAI LLM with another LangChain-supported model with minimal changes.
- Readability: The code is clean, follows Python's PEP 8 conventions, and uses descriptive variable names (data_agent, user_query), making it easy to follow the logic.
- Error Handling: Error handling is robust. The code correctly uses try...except blocks to handle potential FileNotFoundError during data loading and to gracefully catch exceptions during agent invocation, preventing crashes and providing informative error messages to the user.
- Minor Issue: A small copy-paste error was noted where the code for the conversational_agent is duplicated at the bottom of the charting_agent.py file. While this doesn't affect runtime execution (as main.py imports correctly), it is a minor code quality issue that should be cleaned up.

4. Data Sources, Design Choices, Challenges

Data Sources:

- Primary Data Source: A local CSV file, netflix_titles.csv. The path is configured in config.py.
- User Input: Natural language queries provided via the command-line interface.

Design Choices:

- Specialized Charting Agent: This was a key design decision. Creating a separate agent with a highly constrained prompt was necessary to reliably control the LLM's behavior for the specific task of saving plots, rather than just displaying them or describing them.
- Hardcoded Charting Keywords: Using a predefined list of keywords for routing is a simple, fast, and effective choice for this application's scope, avoiding the complexity and potential latency of an LLM-based router.
- matplotlib.use('Agg'): The inclusion of this line is a thoughtful and practical choice, indicating that the developer proactively solved a common real-world challenge where Matplotlib fails in non-GUI environments.
- allow_dangerous_code=True: This is a necessary setting for the pandas agent to execute its generated code. This was a required choice, acknowledging the inherent security trade-offs of such agents.

Challenges Faced:

- Controlling Agent Output: The extremely detailed prompt in query_charting_agent strongly suggests the developer faced challenges in forcing the agent to conform to a strict output format (i.e., save a file, don't use plt.show(), and only return the path). The final prompt is a testament to successful iterative prompt engineering to overcome this.
- Environment-Specific Errors: The presence of matplotlib.use('Agg') indicates the developer likely encountered and debugged RuntimeError exceptions related to Matplotlib's backend when running the script in a terminal or server environment.

- Ensuring Response Quality: The creation of the conversational_agent implies that the developer recognized the raw output from the primary agent was not sufficiently user-friendly and proactively designed a solution to enhance the final deliverable.