AI AGENT-POWERED AUTOMATION FOR PELOTON FITNESS ECOSYSTEM: DESIGN & PROTOTYPE PHASE

Kevin Geidel
MSDS 442: AI Agent Design & Development
Northwestern University
June 1, 2025

Requirement 1: Directory structure and training data

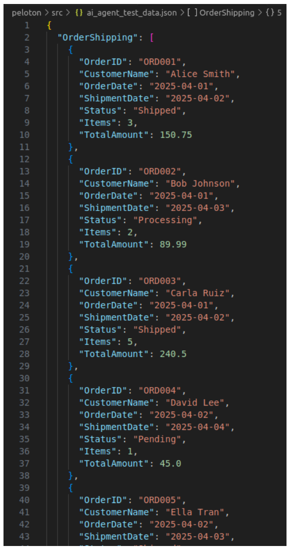
There are a number of files involved in prototyping the Peloton automation AI agent. A full breakdown of the project's code base can be found in figure 1. `Analysis.pdf` is this document. `requirements.txt` is a list of all the Python packages needed to run the agent. They can be installed using `pip` and a Python virtual environment manager (such as `pyenv`.) There are some instructions for cloning the repository in order to run the agent on your local machine in the `README`.

```
msds442/
├── Analysis.pdf
├── requirements.txt
├── README.md
└── peloton/
    ├── src/
    │   ├── ai_agent_test_data.json
    │   └── peloton_data_chatgpt_raw_responses.ipynb
    ├── __init__.py
    ├── agents.py
    └── Phase_2_NOTEBOOK_Geidel.ipynb
```

Figure 1: Directory structure for Peloton AI Agent

Inside the `peloton/` directory we see the `src/` directory. This contains the agent data used for testing. The test data was generated (in phase 1) through a series of prompts to ChatGPT (`https://chatgpt.com/`). The prompts and raw outputs are stored in `peloton_data_chatgpt_raw_responses.ipynb`. The generated test data covers the five AI agents (marketing, data science, membership & fraud detection, orders and product recommendations.) Each agent has test data that supports at least three specific user stories. The records are stored, in JSON format, in `ai_agent_test_data.json`. A sample of what these JSON objects look like can be seen in figure 2

Back in the `peloton/` directory we see `__init__.py`. This file makes the `peloton/` directory recognizable as a Python module to the interpreter and allows us to import it as such. The actual `PelotonAgent` class is defined in `agents.py` and will be described in further detail below (requirement 3.) The Jupyter notebook file, `Phase_2_NOTEBOOK_Geidel.ipynb`, is used for the demonstration. It simply imports the agent class, instantiates an instance, renders a graphic that depicts the LangGraph and invokes the agent.


peloton > src > {} ai_agent_test_data.json > [ ] OrderShipping > {} 5

```
{
  "OrderShipping": [
    {
      "OrderID": "ORD001",
      "CustomerName": "Alice Smith",
      "OrderDate": "2025-04-01",
      "ShipmentDate": "2025-04-02",
      "Status": "Shipped",
      "Items": 3,
      "TotalAmount": 150.75
    },
    {
      "OrderID": "ORD002",
      "CustomerName": "Bob Johnson",
      "OrderDate": "2025-04-01",
      "ShipmentDate": "2025-04-03",
      "Status": "Processing",
      "Items": 2,
      "TotalAmount": 89.99
    },
    {
      "OrderID": "ORD003",
      "CustomerName": "Carla Ruiz",
      "OrderDate": "2025-04-01",
      "ShipmentDate": "2025-04-02",
      "Status": "Shipped",
      "Items": 5,
      "TotalAmount": 240.5
    },
    {
      "OrderID": "ORD004",
      "CustomerName": "David Lee",
      "OrderDate": "2025-04-02",
      "ShipmentDate": "2025-04-04",
      "Status": "Pending",
      "Items": 1,
      "TotalAmount": 45.0
    },
    {
      "OrderID": "ORD005",
      "CustomerName": "Ella Tran",
      "OrderDate": "2025-04-02",
      "ShipmentDate": "2025-04-03",
```

Figure 2: A sample of the test data in JSON format.

Requirement 2: LangGraph architecture

The actual agent graph is created (using `LangGraph`) in the `build_graph` method of the `PelotonAgent` class (see lines 116-143 in the attached listing.) The nodes and their respective functions are defined above this. Note the use of conditional edges to allow for agent choice and the inclusion of a `ToolNode` that houses the LangChain retriever. The result of integrating these agents into this workflow can be seen graphically in figure 3. Dotted lines indicate conditional edges.

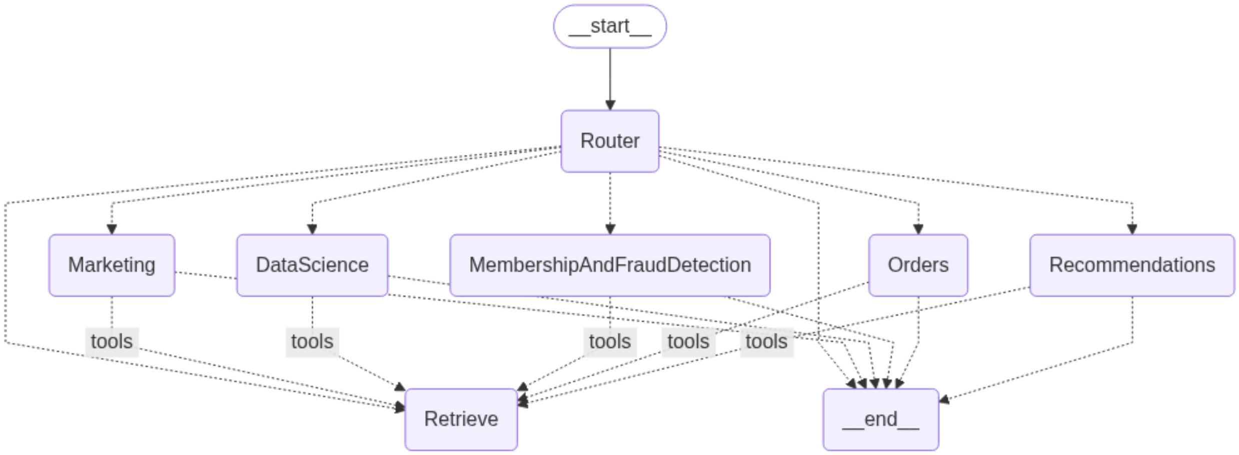


Figure 3: The Peloton agent architecture.

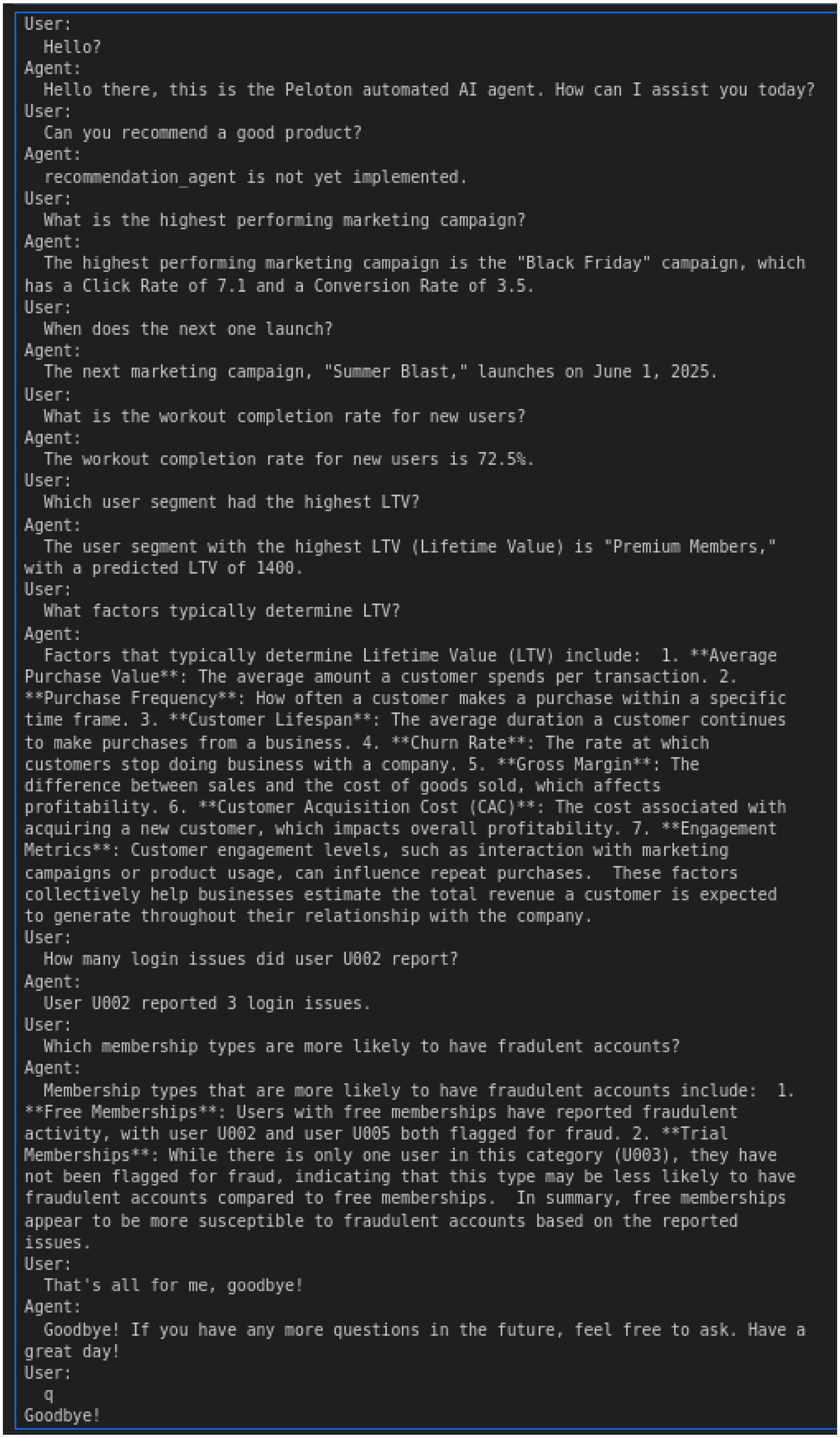
Requirement 3: Implementation

Three area specific agents (plus the router agent) are implemented in this phase (marketing, data science and membership/fraud detection.) In order to discuss their implementation we must examine the `PelotonAgent` class defined in `agents.py`. The contents of this file are included in the attached listing. `LangGraph` is a stateful machine and we give our state attributes in lines 47-52. Each node will track the current inquiry, the latest response, the node that referred the inquiry to the current node and the name of the next node to forward to. Messages are included to provide memory in the form of historical context. Upon instantiating an object of this class we create an LLM client, load the test data into LangChain documents and store these documents in a ChromaDb vector store (lines 61, 64 and 67 respectively.) `load_documents` and `populate_vector_store` and class methods defined below. There are several additional worker methods defined on the class that perform utility functions- mostly to abstract/DRY-out the code. The agent definitions themselves begin on line 201 with the `router_agent`. The core function of the router is to classify the user's intent and set `next_node` accordingly.

The implemented agents share similar patterns. They are centered around a system prompt that defines their purpose and queries the LLM on behalf of the user. All messages, AI, System and Human are added to the messages list to maintain a memory of the conversation over multiple iterations.

Requirement 4: Working prototype

Three user stories (one for each of the implemented agents) were tested. All three agents are able to access their respective data from the store. The marketing agent was developed so that it might extract analyze the performance of ongoing marketing campaigns. The data science agent is able to derive insights from metrics in the data store. The membership agent can answer questions about potential fraudulent accounts. The following transcript of a session tried to demonstrate all of the implemented features: routing, accessing data, drawing insights and access to general knowledge from LLM training.



```
User:
  Hello?
Agent:
  Hello there, this is the Peloton automated AI agent. How can I assist you today?
User:
  Can you recommend a good product?
Agent:
  recommendation_agent is not yet implemented.
User:
  What is the highest performing marketing campaign?
Agent:
  The highest performing marketing campaign is the "Black Friday" campaign, which
has a Click Rate of 7.1 and a Conversion Rate of 3.5.
User:
  When does the next one launch?
Agent:
  The next marketing campaign, "Summer Blast," launches on June 1, 2025.
User:
  What is the workout completion rate for new users?
Agent:
  The workout completion rate for new users is 72.5%.
User:
  Which user segment had the highest LTV?
Agent:
  The user segment with the highest LTV (Lifetime Value) is "Premium Members,"
with a predicted LTV of 1400.
User:
  What factors typically determine LTV?
Agent:
  Factors that typically determine Lifetime Value (LTV) include:  1. **Average
Purchase Value**: The average amount a customer spends per transaction. 2.
**Purchase Frequency**: How often a customer makes a purchase within a specific
time frame. 3. **Customer Lifespan**: The average duration a customer continues
to make purchases from a business. 4. **Churn Rate**: The rate at which
customers stop doing business with a company. 5. **Gross Margin**: The
difference between sales and the cost of goods sold, which affects
profitability. 6. **Customer Acquisition Cost (CAC)**: The cost associated with
acquiring a new customer, which impacts overall profitability. 7. **Engagement
Metrics**: Customer engagement levels, such as interaction with marketing
campaigns or product usage, can influence repeat purchases.  These factors
collectively help businesses estimate the total revenue a customer is expected
to generate throughout their relationship with the company.
User:
  How many login issues did user U002 report?
Agent:
  User U002 reported 3 login issues.
User:
  Which membership types are more likely to have fradulent accounts?
Agent:
  Membership types that are more likely to have fraudulent accounts include:  1.
**Free Memberships**: Users with free memberships have reported fraudulent
activity, with user U002 and user U005 both flagged for fraud. 2. **Trial
Memberships**: While there is only one user in this category (U003), they have
not been flagged for fraud, indicating that this type may be less likely to have
fraudulent accounts compared to free memberships.  In summary, free memberships
appear to be more susceptible to fraudulent accounts based on the reported
issues.
User:
  That's all for me, goodbye!
Agent:
  Goodbye! If you have any more questions in the future, feel free to ask. Have a
great day!
User:
  q
Goodbye!
```

Figure 4: Example session that jogs three user stories and other features.

```python
########################################################################
# MSDS 442: AI Agent Design and Development
# Spring ’25
# Dr. Bader
#
# Final Project: AI Agent Automation for Peloton’s Fitness Ecosystem
# Phase 2 - Prototype
#
# Kevin Geidel
#
########################################################################

# OBJECTIVE:
#   Construct a high-fidelity prototype of the Peloton Automation.
#   Implement the planned architecture using Phase 1 Artifacts.

# Load environment variables
from dotenv import load_dotenv
load_dotenv()

# Python native imports
import os, inspect, textwrap, time, sys
from typing import Annotated, Sequence

# LangChain/LangGraph imports
from langchain_core.messages import BaseMessage, HumanMessage, SystemMessage
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_community.document_loaders import JSONLoader
from langchain.tools.retriever import create_retriever_tool
from langchain.embeddings.sentence_transformer import SentenceTransformerEmbeddings
os.environ[’USER_AGENT’] = ’Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3’
__import__(’pysqlite3’)
sys.modules[’sqlite3’] = sys.modules.pop(’pysqlite3’)
from langchain_community.vectorstores import Chroma
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver
from langgraph.prebuilt import ToolNode, tools_condition

# 3rd party package imports
from IPython.display import display, Image
from typing_extensions import TypedDict


class PelotonAgent:
    ’’’ Namespace for methods and metaclasses that facilliate Peloton’s Agent-based automation. ’’’

    class InquiryState(TypedDict):
        inquiry: str
        response: str
        referring_node: str
        next_node: str
        messages: Annotated[Sequence[BaseMessage], "List of messages in the conversation"]

    def __init__(self):
        # Assign agent-wide variables
        self.model_name = 'gpt-4o-mini'
        self.data_dir = os.path.join('src')
        self.agent_data_path = os.path.join(self.data_dir, 'ai_agent_test_data.json')

        # Establish the AI client
        self.llm = ChatOpenAI(model=self.model_name, temperature=0)

        # Load test data into memory in form of langchain docs
        self.load_documents()

        # Initialize ChromaDb vector store and load docs
        self.populate_vector_store()

        # Construct the agent graph
        self.build_graph()

    def load_documents(self):
        loader = JSONLoader(
            file_path=self.agent_data_path,
            jq_schema='.',
            text_content=False,
        )
        self.data= loader.load()

    def populate_vector_store(self):
        if not hasattr(self, 'db'):
            self.db = Chroma.from_documents(
                documents=self.data,
                collection_name='test_data',
                embedding=OpenAIEmbeddings()
            )

    def get_retriever_tool(self):
        return create_retriever_tool(
            self.db.as_retriever(),
            'retrieve_peloton_data',
            """Search Peloton Enterprise data in the vector store and return information for:
                - Order and Shipping information
                - Product catalog information
                - Marketing campaign metrics
                - Membership account information
                - Data Science metrics"""
        )

    def extract_from_state(self, state):
        inquiry = state.get('inquiry', '')
        messages = state.get("messages", [])
        history = "\n".join([f'{msg.type}: {msg.content}' for msg in messages][:5])
        return inquiry, messages, history

```

```python
    def get_standard_human_message(self, inquiry, history):
        return HumanMessage(
            content = f"""Provide an answer for the following user's inquiry:
            '{inquiry}'

            Conversation history for content:
            {history}
            """.strip()
        )

    def build_graph(self):
        builder = StateGraph(self.InquiryState)
        # nodes
        builder.add_node('Router', self.router_agent)
        builder.add_node('Marketing', self.marketing_agent)
        builder.add_node('DataScience', self.data_science_agent)
        builder.add_node('MembershipAndFraudDetection', self.membership_agent)
        builder.add_node('Orders', self.orders_agent)
        builder.add_node('Recommendations', self.recommendation_agent)
        retriever_tool = ToolNode([self.get_retriever_tool()])
        builder.add_node("Retrieve", retriever_tool)
        # edges/workflow
        builder.add_edge(START, 'Router')
        builder.add_conditional_edges(
            'Router',
            lambda x: x['next_node'],
        )
        for node in ['Marketing', 'DataScience', 'MembershipAndFraudDetection', 'Orders', 'Recommendations']:
            builder.add_conditional_edges(
                node,
                tools_condition,
                {
                    'tools': 'Retrieve',
                    END: END,
                }
            )

        self.graph = builder.compile(checkpointer=MemorySaver())

    def draw_graph(self):
        display(Image(self.graph.get_graph().draw_mermaid_png()))

    # base methods for agents
    def termination_check(self, state):
        ''' Check for user end session '''
        inquiry = state.get('inquiry', '')
        if inquiry.lower() in ['q', 'quit', 'goodbye', 'bye']:
            return {
                "inquiry": inquiry,
                "referring_node": state.get('next_node', 'Router'),
                "next_node": END,
                "response": "Goodbye! Thank you for contacting the Peloton automated AI agent!",
                "messages": state.get('messages', []) + [HumanMessage(content=inquiry), SystemMessage(content="Conversation ended by user.")]
            }
        else:
            return None

    def route_ongoing_chat(self, state, max_history=5):
        inquiry = state.get('inquiry', '')
        messages = state.get("messages", [])
        if state.get('referring_node') != "Router" and state.get('next_node'):
            history = "\n".join([f"{msg.type}: {msg.content}" for msg in state.get("messages", [])][:max_history])
            query = f"""Given the conversation history and the new inquiry: '{inquiry}', determine if this is a follow-up question related to the previous department ({state['referring_node']}) or a new topic. Return 'continue' if it's a follow-up, or classify the intent for a new topic.
            Possible intent values: Greeting, GeneralInquiry, Marketing, DataScience, MembershipAndFraudDetection, Orders, Recommendations

            Conversation history:
            {history}
            """
            messages_for_intent = [
                SystemMessage(content="You are a helpful assistant tasked with classifying the intent of a user's query or detecting follow-ups."),
                HumanMessage(content=[{'type': 'text', 'text': query}])
            ]
            response = self.llm.invoke(messages_for_intent)
            intent = response.content.strip()
            if intent == 'continue':
                return {
                    "inquiry": state["inquiry"],
                    "referring_node": "Router",
                    "next_node": state['referring_node'],
                    "response": f"Routing to the {state['referring_node']} department.",
                    "messages": messages + [HumanMessage(content=inquiry)]
                }
        return {}

    def unimplemented_agent(self, state):
        calling_agent = inspect.currentframe().f_back.f_code.co_name
        return {
            'inquiry': state['inquiry'],
            'response': f'{calling_agent} is not yet implemented.',
            'referring_node': state.get('referring_node', None),
            'next_node': END,
            'messages': state.get('messages', []) + [SystemMessage(content=f'Routed to unimplented agent, {calling_agent}.')]
        }

    # define agents methods
    def router_agent(self, state):
        inquiry = state.get('inquiry', '')
        messages = state.get('messages', [])

        # check for termination by user
        terminate = self.termination_check(state)
        if terminate:
            return terminate

        # check for ongoing conversation
        ongoing = self.route_ongoing_chat(state)
        if ongoing:
            return ongoing

        # Classify intent for this new session and route
        query = f"""Classify the user’s intents based on the following input: ’{inquiry}’.
                List of possible intent values: Greeting, GeneralInquiry, Marketing, DataScience, MembershipAndFraudDetection, Orders, Recommendations
                Questions about user accounts or login issues goto MembershipAndFraudDetection
                Return only the intent value of the inquiry identified with no extra text or characters"""
        messages = [
            SystemMessage(content="You are a helpful assistant tasked with classifying the intent of user’s inquiry"),
            HumanMessage(content=[{"type": "text", "text": query}]),
        ]
        response = self.llm.invoke(messages)
        intent = response.content.strip()
        response_lower = intent.lower()

        if "greeting" in response_lower:
            response = "Hello there, this is the Peloton automated AI agent. How can I assist you today?"
            next_node = END
        elif "generalinquiry" in response_lower:
            response = "For general informtion about Peloton’s ecosystem of classes and products visit https://www.onepeloton.com/. Thank you!"
            next_node = END
        else:
            response = f"Let me forward your query to our {intent} agent."
            next_node = intent

        return {
            "inquiry": state["inquiry"],
            "referring_node": "Router",
            "next_node": next_node,
            "response": response,
            ’messages’: messages + [SystemMessage(content=intent)]
        }

    def marketing_agent(self, state):
        # Target use cases:
        #   1) Query customer data via vector DB; use retrieval to analyze and return top segments.
        inquiry, messages, history = self.extract_from_state(state)
        marketing_agent_human_message = HumanMessage(
            content = f"""Provide an answer for the following user’s inquiry:
            ’{inquiry}’

            Conversation history for content:
            {history}
            """.strip()
        )
        if state['referring_node'] == 'Router':
            marketing_agent_system_message = SystemMessage(
                content = f"""You are a helpful assistant tasked with retrieving and organizing data to answer questions about ongoing marketing campaigns.
                If the inquiry relates to data found in the agent database, base answers solely on the records within: {str(self.data)}"""
            )
            messages += [marketing_agent_system_message, marketing_agent_human_message]
        else:
            messages += [marketing_agent_human_message]

        # query the llm
        response = self.llm.invoke(messages)

        return {
            'inquiry': inquiry,
            'referring_node': 'Marketing',
            'next_node': tools_condition(state),
            'response': response,
            'messages': messages + [SystemMessage(content=response.content.strip())]
        }

    def data_science_agent(self, state):
        # Target use cases:
        #   1) Analyze trends by user segment.
        inquiry, messages, history = self.extract_from_state(state)
        if state['referring_node'] == 'Router':
            marketing_agent_system_message = SystemMessage(
                content = f"""You are a helpful assistant tasked with performing data science and analytics.
                If the inquiry relates to data found in the agent database, base answers solely on the records within: {str(self.data)}"""
            )
            messages += [marketing_agent_system_message, self.get_standard_human_message(inquiry, history)]
        else:
            messages += [self.get_standard_human_message(inquiry, history)]

        # query the llm
        response = self.llm.invoke(messages)

        return {
            'inquiry': inquiry,
            'referring_node': 'DataScience',
            'next_node': tools_condition(state),
            'response': response,
            'messages': messages + [SystemMessage(content=response.content.strip())]
        }

    def membership_agent(self, state):
        # Target use cases:
        #   1) Analyze login patterns to aid in fraud detection.
        inquiry, messages, history = self.extract_from_state(state)
        if state['referring_node'] == 'Router':
            marketing_agent_system_message = SystemMessage(
                content = f"""You are a helpful assistant tasked with answering questions about membership accounts and detecting fradulent login attempts.
                If the inquiry relates to data found in the agent database, base answers solely on the records within: {str(self.data)}"""
            )
            messages += [marketing_agent_system_message, self.get_standard_human_message(inquiry, history)]
        else:
            messages += [self.get_standard_human_message(inquiry, history)]

        # query the llm
        response = self.llm.invoke(messages)

        return {
            'inquiry': inquiry,
            'referring_node': 'MembershipAndFraudDetection',
            'next_node': tools_condition(state),
            'response': response,
            'messages': messages + [SystemMessage(content=response.content.strip())]
        }

    def orders_agent(self, state):
        return self.unimplemented_agent(state)

    def recommendation_agent(self, state):
        return self.unimplemented_agent(state)

    def invoke(self, thread_id="1"):
        config = {"configurable": {"thread_id": thread_id}}
        while True:
            user_input = input("User: ")
            time.sleep(0.5)
            print(f"User:\n  {user_input}")
            time.sleep(0.5)
            if user_input.lower() in {"q", "quit"}:
                print("Goodbye!")
                break
            result = self.graph.invoke({"inquiry": user_input}, config=config)
            time.sleep(0.5)
            response = result.get("response", "No Response Returned")
            if not isinstance(response, str):
                response = response.content
            print('Agent:\n ', textwrap.fill(response, 80))
```