project

March 15, 2025

1 Telecom Service Assistant - Final Project

1.1 Project Overview

In this project, you will build a comprehensive telecom service assistant that integrates multiple AI frameworks:

- **LangGraph**: Orchestration layer that coordinates the flow between different components
- **CrewAI**: Specialized customer support for billing and account queries
- **AutoGen**: Network troubleshooting with multiple specialized agents
- **LangChain**: Service recommendations using ReAct agents
- **LlamaIndex**: Knowledge retrieval from documentation

This project brings together all the concepts you've learned throughout the course!

1.2 Project Structure

We'll follow a modular architecture to organize our code. Here's the recommended file structure:

```
telecom_assistant/
  app.py                        # Main entry point to run the Streamlit app
  requirements.txt              # Project dependencies
  data/
      telecom.db                # SQLite database for telecom data
      documents/                # Folder for knowledge base documents
          service_plans.md
          network_guide.md
          billing_faq.md
          technical_support.md
  config/
      config.py                 # Configuration settings
  agents/
      __init__.py
      billing_agents.py         # CrewAI implementation
      network_agents.py         # AutoGen implementation
      service_agents.py         # LangChain implementation
      knowledge_agents.py       # LlamaIndex implementation
  orchestration/
      __init__.py
      graph.py                  # LangGraph orchestration
```

```
    state.py                # State management
ui/
    __init__.py
    streamlit_app.py        # Streamlit UI code
utils/
    __init__.py
    database.py             # Database utilities
    document_loader.py      # Document loading utilities
```

1.3 1. Database Setup

I have provided you with the required database with sample data also

Also, i have provided all the documents required for storing in vector store in data folder given to you

1.4 Core Concepts and Flow Orchestration

Let's examine how different query types will flow through our application. This understanding is crucial for implementing the system correctly.

1.4.1 Overview of the Query Flow

The general flow for any query in our system follows these steps:

1. **Query Submission**: User submits query through Streamlit interface
2. **Classification**: LangGraph classifies the query type
3. **Routing**: Query is routed to the appropriate specialized framework
4. **Processing**: The specialized framework processes the query
5. **Response Formulation**: Results are formatted into a user-friendly response
6. **Presentation**: Response is presented to the user

Now let's look at each specific query type in detail:

1.4.2 1. Billing and Account Queries (Using CrewAI)

Example Queries: - "Why is my bill higher this month?" - "I want to understand charges on my last statement" - "What add-on packs are available for my plan?"

Flow in Detail:

1. User submits a billing query via Streamlit
2. Query is sent to the LangGraph orchestrator
3. The `classify_query` node determines this is a billing/account query
4. The query is routed to the `crew_ai_node`
5. In the CrewAI node:
    - A Billing Specialist agent analyzes the bill components
    - A Service Advisor agent examines plan details and changes
    - The agents collaborate to generate a comprehensive explanation
6. The response returns to the `formulate_response` node
7. The final answer is presented to the user

Key Implementation Points: - Use CrewAI's Agent class with specialized roles - Configure agents with SQLDatabaseTool access - Create tasks that break down the billing analysis process

1.4.3 2. Network Troubleshooting Queries (Using AutoGen)

Example Queries: - "I can't make calls from my home area" - "My data connection keeps dropping" - "Why is my 5G speed so slow?"

Flow in Detail:

1. User submits a network issue query via Streamlit
2. Query is sent to the LangGraph orchestrator
3. The `classify_query` node identifies this as a network troubleshooting query
4. The query is routed to the `autogen_node`
5. In the AutoGen node:
   - A NetworkDiagnosticsAgent checks for known issues and patterns
   - A RouterConfigurationAgent examines technical parameters
   - A ConnectivitySpecialistAgent prepares troubleshooting steps
   - The GroupChat allows the agents to collaborate
6. The collective solution returns to the `formulate_response` node
7. The final answer is presented to the user

Key Implementation Points: - Use AutoGen's AssistantAgent with specialized roles - Set up GroupChat for agent collaboration - Configure agents with both database and vector store access

1.4.4 3. Service Recommendation Queries (Using LangChain)

Example Queries: - "What's the best plan for a family of four?" - "I need a plan with international roaming" - "Which plan is best for video streaming?"

Flow in Detail:

1. User submits a plan recommendation query via Streamlit
2. Query is sent to the LangGraph orchestrator
3. The `classify_query` node identifies this as a service recommendation query
4. The query is routed to the `langchain_node`
5. In the LangChain node:
   - A ReAct agent analyzes the user's requirements
   - It queries the database for plan information
   - It applies reasoning to match requirements to features
   - It generates a personalized recommendation
6. The recommendation returns to the `formulate_response` node
7. The final answer is presented to the user

Key Implementation Points: - Use LangChain's create_react_agent with appropriate tools - Implement SQLDatabaseTool for plan information retrieval - Create a custom prompt for plan recommendations

1.4.5 4. Technical Documentation Queries (Using LlamaIndex)

Example Queries: - "How do I enable VoLTE on my phone?" - "What are the APN settings for Android?" - "Tell me about 5G coverage in Mumbai"

Flow in Detail:

1. User submits a technical documentation query via Streamlit
2. Query is sent to the LangGraph orchestrator
3. The `classify_query` node identifies this as a knowledge retrieval query
4. The query is routed to the `llamaindex_node`
5. In the LlamaIndex node:
   - A RouterQueryEngine determines the best source for the information
   - It could use vector search for conceptual questions
   - It could use SQL for factual data (like coverage areas)
   - It might use a HybridQueryEngine to combine both approaches
6. The retrieved information returns to the `formulate_response` node
7. The final answer is presented to the user

Key Implementation Points: - Use LlamaIndex's document processing to create vector embeddings - Set up RouterQueryEngine to select the appropriate backend - Implement hybrid search capabilities when needed

1.5 LangGraph Orchestration Implementation

The central component of our system is the LangGraph orchestrator. Here's a template to get you started:

[ ]:
```python
# orchestration/graph.py
from typing import TypedDict, Dict, Any, List
from langgraph.graph import StateGraph, END

# Define the state structure
class TelecomAssistantState(TypedDict):
    query: str                              # The user's original query
    customer_info: Dict[str, Any]           # Customer information if available
    classification: str                     # Query classification
    intermediate_responses: Dict[str, Any] # Responses from different nodes
    final_response: str                     # Final formatted response
    chat_history: List[Dict[str, str]]      # Conversation history

# Classification node - determines query type
def classify_query(state: TelecomAssistantState) -> TelecomAssistantState:
    """Classify the query into different categories"""
    # TODO: Implement query classification logic
    # Hint: Use an LLM to classify the query into one of these categories:
    # - billing_account
    # - network_troubleshooting
    # - service_recommendation
    # - knowledge_retrieval

    # For testing, you can hardcode classifications based on keywords
    query = state["query"].lower()
```

```
    # Sample classification logic Replace this with llm call
    classification = "billing_account"  # Default

    if any(word in query for word in ["bill", "charge", "payment", "account"]):
        classification = "billing_account"
    elif any(word in query for word in ["network", "signal", "connection",␣
 ↪"call", "data", "slow"]):
        classification = "network_troubleshooting"
    elif any(word in query for word in ["plan", "recommend", "best", "upgrade",␣
 ↪"family"]):
        classification = "service_recommendation"
    elif any(word in query for word in ["how", "what", "configure", "setup",␣
 ↪"apn", "volte"]):
        classification = "knowledge_retrieval"

    return {**state, "classification": classification}

# Routing function - determines next node based on classification
def route_query(state: TelecomAssistantState) -> str:
    """Route the query to the appropriate node based on classification"""
    # TODO: Implement query routing logic
    # TODO: Check the classification in the state
    # TODO: Return the appropriate node name based on classification:
    #   - For "billing_account" classification, return "crew_ai_node"
    #   - For "network_troubleshooting" classification, return "autogen_node"
    #   - For "service_recommendation" classification, return "langchain_node"
    #   - For "knowledge_retrieval" classification, return "llamaindex_node"
    #   - For any other classification, return "fallback_handler"

    # TODO: Remove this placeholder return statement when implementing
    return "fallback_handler"  # Default placeholder return

# Node function templates for each framework
def crew_ai_node(state: TelecomAssistantState) -> TelecomAssistantState:
    """Handle billing and account queries using CrewAI"""
    # TODO: Implement CrewAI processing
    # Hint: Import and use your CrewAI implementation from billing_agents.py

    # For testing, return a placeholder response
    response = f"This would be processed by CrewAI: {state['query']}"
    return {**state, "intermediate_responses": {"crew_ai": response}}

def autogen_node(state: TelecomAssistantState) -> TelecomAssistantState:
    """Handle network troubleshooting using AutoGen"""
    # TODO: Implement AutoGen processing
    # Hint: Import and use your AutoGen implementation from network_agents.py
```

```
    # For testing, return a placeholder response
    response = f"This would be processed by AutoGen: {state['query']}"
    return {**state, "intermediate_responses": {"autogen": response}}

def langchain_node(state: TelecomAssistantState) -> TelecomAssistantState:
    """Handle service recommendations using LangChain"""
    # TODO: Implement LangChain processing
    # Hint: Import and use your LangChain implementation from service_agents.py

    # For testing, return a placeholder response
    response = f"This would be processed by LangChain: {state['query']}"
    return {**state, "intermediate_responses": {"langchain": response}}

def llamaindex_node(state: TelecomAssistantState) -> TelecomAssistantState:
    """Handle knowledge retrieval using LlamaIndex"""
    # TODO: Implement LlamaIndex processing
    # Hint: Import and use your LlamaIndex implementation from knowledge_agents.
 ↪py

    # For testing, return a placeholder response
    response = f"This would be processed by LlamaIndex: {state['query']}"
    return {**state, "intermediate_responses": {"llamaindex": response}}

def fallback_handler(state: TelecomAssistantState) -> TelecomAssistantState:
    """Handle queries that don't fit other categories"""
    response = "I'm not sure how to help with that specific question. Could you␣
 ↪try rephrasing or ask about our services, billing, network issues, or␣
 ↪technical support?"
    return {**state, "intermediate_responses": {"fallback": response}}

def formulate_response(state: TelecomAssistantState) -> TelecomAssistantState:
    """Create final response from intermediate results"""
    # Get the response from whichever node was called
    intermediate_responses = state["intermediate_responses"]

    # Extract the first response we find (in a real implementation, you'd␣
 ↪format this better)
    response_value = next(iter(intermediate_responses.values()))

    return {**state, "final_response": response_value}

def create_graph():
    """Create and return the workflow graph"""
    # Build the graph
    workflow = StateGraph(TelecomAssistantState)

    # Add nodes
```

```
    workflow.add_node("classify_query", classify_query)
    workflow.add_node("crew_ai_node", crew_ai_node)
    workflow.add_node("autogen_node", autogen_node)
    workflow.add_node("langchain_node", langchain_node)
    workflow.add_node("llamaindex_node", llamaindex_node)
    workflow.add_node("fallback_handler", fallback_handler)
    workflow.add_node("formulate_response", formulate_response)

    # Add conditional edges from classification to appropriate node
    workflow.add_conditional_edges(
        "classify_query",
        route_query,
        {
            "crew_ai_node": "crew_ai_node",
            "autogen_node": "autogen_node",
            "langchain_node": "langchain_node",
            "llamaindex_node": "llamaindex_node",
            "fallback_handler": "fallback_handler"
        }
    )

    # Add edges from each processing node to response formulation
    workflow.add_edge("crew_ai_node", "formulate_response")
    workflow.add_edge("autogen_node", "formulate_response")
    workflow.add_edge("langchain_node", "formulate_response")
    workflow.add_edge("llamaindex_node", "formulate_response")
    workflow.add_edge("fallback_handler", "formulate_response")
    workflow.add_edge("formulate_response", END)

    # Set the entry point
    workflow.set_entry_point("classify_query")

    # Compile the graph
    return workflow.compile()
```

1.6 Streamlit Frontend Implementation

Here's the complete Streamlit UI implementation that you can use for your project:

[ ]:
```
# ui/streamlit_app.py
import streamlit as st
import pandas as pd
import os
import sys
from pathlib import Path

# Add parent directory to path so we can import from other modules
```

```
sys.path.append(str(Path(__file__).parent.parent))

# Import core functionality
from orchestration.graph import create_graph, TelecomAssistantState

# Set page configuration
st.set_page_config(
    page_title="Telecom Service Assistant",
    page_icon=" ",
    layout="wide"
)

# Initialize session state variables
if "authenticated" not in st.session_state:
    st.session_state.authenticated = False
if "user_type" not in st.session_state:
    st.session_state.user_type = None
if "email" not in st.session_state:
    st.session_state.email = None
if "chat_history" not in st.session_state:
    st.session_state.chat_history = []
if "graph" not in st.session_state:
    # Initialize the LangGraph workflow
    st.session_state.graph = create_graph()

# Function to process user queries
def process_query(query: str):
    """Process a user query through the LangGraph workflow"""
    # Create initial state
    state = {
        "query": query,
        "customer_info": {"email": st.session_state.email},
        "classification": "",
        "intermediate_responses": {},
        "final_response": "",
        "chat_history": st.session_state.chat_history
    }

    # Process through the graph
    # In a real implementation, you'd use:
    # result = st.session_state.graph.invoke(state)
    # return result["final_response"]

    # For development, simulate responses based on query content
    query_lower = query.lower()
```

```
    if any(word in query_lower for word in ["bill", "charge", "payment",␣
↪"account"]):
        return "Based on your billing inquiry, I can see that your recent bill␣
↪increased due to the addition of an international roaming package ($10) and␣
↪premium content access ($5). Your base plan remains unchanged at $49.99."
    elif any(word in query_lower for word in ["network", "signal",␣
↪"connection", "call", "data", "slow"]):
        return "I've checked our network status, and there's scheduled␣
↪maintenance in your area that might be affecting your connectivity. This␣
↪should be resolved by 6 PM today. In the meantime, try switching to 3G mode␣
↪in your phone settings for more stable connectivity."
    elif any(word in query_lower for word in ["plan", "recommend", "best",␣
↪"upgrade", "family"]):
        return "For a family of four with heavy streaming usage, I recommend␣
↪our Family Share Plus plan at $89.99/month. This includes 40GB of shared␣
↪high-speed data, unlimited talk and text for all lines, and free access to␣
↪our premium streaming service."
    elif any(word in query_lower for word in ["how", "what", "configure",␣
↪"setup", "apn", "volte"]):
        return "To enable VoLTE on your Android device, go to Settings >␣
↪Connections > Mobile Networks > VoLTE calls and toggle it on. Make sure your␣
↪device is VoLTE compatible and you're in a coverage area. This will improve␣
↪call quality and allow simultaneous voice and data usage."
    else:
        return "I'm not sure how to help with that specific question. Could you␣
↪try rephrasing or ask about our services, billing, network issues, or␣
↪technical support?"

# Sidebar for authentication
with st.sidebar:
    st.title("Telecom Service Assistant")

    if not st.session_state.authenticated:
        st.subheader("Login")
        email = st.text_input("Email Address")
        user_type = st.selectbox("User Type", ["Customer", "Admin"])

        if st.button("Login"):
            if email and "@" in email:
                st.session_state.authenticated = True
                st.session_state.user_type = user_type
                st.session_state.email = email
                st.success(f"Logged in as {user_type}")
                st.rerun()
            else:
                st.error("Please enter a valid email address")
```

```
    else:
        st.success(f"Logged in as {st.session_state.user_type}")
        st.text(f"Email: {st.session_state.email}")

        if st.button("Logout"):
            st.session_state.authenticated = False
            st.session_state.user_type = None
            st.session_state.email = None
            st.session_state.chat_history = []
            st.rerun()

# Main app content
if st.session_state.authenticated:
    if st.session_state.user_type == "Customer":
        st.title("Welcome to Telecom Service Assistant")

        tab1, tab2, tab3 = st.tabs(["Chat Assistant", "My Account", "Network␣
 ↪Status"])

        with tab1:
            st.header("Chat with our AI Assistant")

            # Display chat history
            for message in st.session_state.chat_history:
                with st.chat_message(message["role"]):
                    st.write(message["content"])

            # Chat input
            if prompt := st.chat_input("How can I help you today?"):
                # Add user message to chat history
                st.session_state.chat_history.append({"role": "user", "content":
 ↪ prompt})

                # Display user message
                with st.chat_message("user"):
                    st.write(prompt)

                # Process user query through LangGraph
                with st.chat_message("assistant"):
                    with st.spinner("Thinking..."):
                        response = process_query(prompt)
                        st.write(response)

                # Add assistant response to chat history
                st.session_state.chat_history.append({"role": "assistant",␣
 ↪"content": response})
```

```
        with tab2:
            st.header("My Account Information")
            st.subheader("Current Plan")
            st.write("Standard Plan (STD_500)")

            col1, col2, col3 = st.columns(3)
            with col1:
                st.metric("Data Used", "3.5 GB", "2.1 GB remaining")
            with col2:
                st.metric("Voice Minutes", "320 mins", "Unlimited")
            with col3:
                st.metric("SMS Used", "45", "Unlimited")

            st.subheader("Billing Information")
            st.write("Next Bill Date: June 15, 2023")
            st.write("Monthly Charge:  799.00")

        with tab3:
            st.header("Network Status")
            status_df = pd.DataFrame({
                "Region": ["Mumbai", "Delhi", "Bangalore", "Chennai",␣
↪"Hyderabad"],
                "4G Status": ["Normal", "Normal", "Degraded", "Normal",␣
↪"Normal"],
                "5G Status": ["Normal", "Maintenance", "Normal", "Normal",␣
↪"Degraded"]
            })

            st.dataframe(status_df, use_container_width=True)

            st.subheader("Known Issues")
            st.info("Scheduled maintenance in Delhi region (03:00-05:00 AM)")
            st.warning("Network congestion reported in Bangalore South")

    elif st.session_state.user_type == "Admin":
        st.title("Admin Dashboard")

        tab1, tab2, tab3 = st.tabs(["Knowledge Base Management", "Customer␣
↪Support", "Network Monitoring"])

        with tab1:
            st.header("Knowledge Base Management")

            st.subheader("Upload Documents to Knowledge Base")
            uploaded_file = st.file_uploader("Upload PDF, Markdown, or Text␣
↪files",
                                             type=["pdf", "md", "txt"],
```

```
                                        accept_multiple_files=True)

            if uploaded_file:
                for file in uploaded_file:
                    st.success(f"Processed {file.name} and added to knowledge␣
↪base")

            st.subheader("Existing Documents")
            doc_df = pd.DataFrame({
                "Document Name": ["Service Plans Guide.md", "Network␣
↪Troubleshooting Guide.md", "Billing FAQs.md", "Technical Support Guide.md"],
                "Type": ["Markdown", "Markdown", "Markdown", "Markdown"],
                "Last Updated": ["2023-06-20", "2023-06-18", "2023-06-15",␣
↪"2023-06-10"],
            })

            st.dataframe(doc_df, use_container_width=True)

        with tab2:
            st.header("Customer Support Dashboard")

            st.subheader("Active Support Tickets")
            ticket_df = pd.DataFrame({
                "Ticket ID": ["TKT004", "TKT005"],
                "Customer": ["Ananya Singh", "Vikram Reddy"],
                "Issue": ["Account reactivation", "Slow internet speeds"],
                "Status": ["In Progress", "Assigned"],
                "Priority": ["Medium", "Medium"],
                "Created": ["2023-06-15", "2023-06-17"]
            })

            st.dataframe(ticket_df, use_container_width=True)

            col1, col2, col3 = st.columns(3)
            with col1:
                st.metric("Open Tickets", "2", "-3")
            with col2:
                st.metric("Avg. Resolution Time", "4.3 hours", "-0.5")
            with col3:
                st.metric("Customer Satisfaction", "92%", "+3%")

        with tab3:
            st.header("Network Monitoring")

            st.subheader("Active Network Incidents")
            incident_df = pd.DataFrame({
                "Incident ID": ["INC003"],
```

```
                "Type": ["Equipment Failure"],
                "Location": ["Delhi West"],
                "Affected Services": ["Voice, Data, SMS"],
                "Started": ["2023-06-15 08:15:00"],
                "Status": ["In Progress"],
                "Severity": ["Critical"]
            })

            st.dataframe(incident_df, use_container_width=True)

# Function to run the app
def main():
    # Already set up above
    pass

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

1.7 Implementation Templates for Each Framework

1.7.1 CrewAI Implementation (Billing Agents)

[ ]:
```
from crewai import Agent, Task, Crew, Process
from langchain.tools import SQLDatabaseTool
from langchain.chat_models import ChatOpenAI

def create_billing_crew():
    """Create and return a CrewAI crew for handling billing inquiries"""
    # TODO: Create an LLM instance
    # - Initialize a ChatOpenAI instance with model="gpt-3.5-turbo" or "gpt-4"
    # - Set temperature=0.1 for consistent, factual responses

    # TODO: Create database tools
    # - Set up SQLDatabaseTool to access the telecom database
    # - Ensure it can query customer bills, plan details, and usage data

    # TODO: Create the Billing Specialist agent
    # - Role: Expert in interpreting billing details and charges
    # - Backstory: "You are a senior billing analyst with 10 years of␣
 ↪experience in telecom"
    # - Goals: Explain bill components, identify unusual changes, clarify all␣
 ↪charges
    # - Tools: Give access to the SQL database tool
    # Example prompt:
    """
    You are an experienced telecom billing specialist who analyzes customer␣
 ↪bills.
```

```
    Your job is to:
    1. Examine the customer's current and previous bills to identify any changes
    2. Explain each charge in simple language
    3. Identify any unusual or one-time charges
    4. Verify that all charges are consistent with the customer's plan

    Use SQL to retrieve billing information, and be precise about numbers.
    Always start by retrieving the customer's most recent bill, then compare it
    with the previous bill to identify changes.
    """

    # TODO: Create the Service Advisor agent
    # - Role: Expert in matching customer needs with service plans
    # - Backstory: "You help customers optimize their telecom services"
    # - Goals: Identify if customer is on optimal plan, suggest alternatives if␣
 ↪needed
    # - Tools: Give access to the SQL database tool
    # Example prompt:
    """
    You are a telecom service advisor who helps customers optimize their plans.
    Your job is to:
    1. Analyze the customer's usage patterns (data, calls, texts)
    2. Compare their usage with their current plan limits
    3. Identify if they are paying for services they don't use
    4. Suggest better plans if available

    Use SQL to retrieve the customer's usage data and plan details.
    Be specific about potential savings or benefits of your recommendations.
    """

    # TODO: Create tasks for the agents
    # - Task 1: Analyze current bill and identify changes (assigned to Billing␣
 ↪Specialist)
    # - Task 2: Review usage patterns and plan fit (assigned to Service Advisor)
    # - Task 3: Generate comprehensive explanation and recommendations␣
 ↪(collaborative)

    # TODO: Create the crew with agents and tasks
    # - Set process to Process.sequential for step-by-step analysis
    # - Add verbose=True for development/debugging

    # TODO: Return the configured crew
    pass

def process_billing_query(customer_id, query):
    """Process a billing query using the CrewAI crew"""
    # TODO: Create the billing crew using the create_billing_crew function
```

```
    # TODO: Process the query
    # - Initialize crew inputs with customer_id and query
    # - Run crew.kickoff() to execute the analysis workflow
    # - Extract the final response

    # TODO: Format the response for the user
    # - Structure the response with clear sections (Bill Analysis, Plan Review,␣
 ↪Recommendations)
    # - Ensure numbers and percentages are clearly presented

    # TODO: Return the formatted response
    pass
```

1.7.2 AutoGen Implementation (Network Agents)

[ ]:
```
import autogen
from langchain.tools import SQLDatabaseTool
from langchain.vectorstores import Chroma

def create_network_agents():
    """Create and return an AutoGen group chat for network troubleshooting"""
    # TODO: Configure AutoGen settings
    # - Set up config_list with the LLM configuration
    # - Configure temperature=0.2 for somewhat deterministic but flexible␣
 ↪responses

    # TODO: Create tools for the agents
    # - Set up tools to query network status database
    # - Create tools to access technical documentation (via vector store)
    # - Add tools to retrieve device-specific troubleshooting information

    # TODO: Create the UserProxyAgent
    # - Configure to receive the initial query
    # - Set up to facilitate the conversation flow
    # Example system message:
    """
    You represent a customer with a network issue. Your job is to:
    1. Present the customer's problem clearly
    2. Ask clarifying questions if agents need more information
    3. Summarize the final solution in simple terms
    """

    # TODO: Create the NetworkDiagnosticsAgent
    # - Focus on network status and outage detection
    # - Give access to network status database
    # Example system message:
```

```
    """
    You are a network diagnostics expert who analyzes connectivity issues.
    Your responsibilities:
    1. Check for known outages or incidents in the customer's area
    2. Analyze network performance metrics
    3. Identify patterns that indicate specific network problems
    4. Determine if the issue is widespread or localized to the customer

    Always begin by checking the network status database for outages in the
    customer's region before suggesting device-specific solutions.
    """

    # TODO: Create the DeviceExpertAgent
    # - Focus on device-specific troubleshooting
    # - Give access to technical documentation
    # Example system message:
    """
    You are a device troubleshooting expert who knows how to resolve
    connectivity issues on different phones and devices.
    Your responsibilities:
    1. Suggest device-specific settings to check
    2. Provide step-by-step instructions for configuration
    3. Explain how to diagnose hardware vs. software issues
    4. Recommend specific actions based on the device type

    Always ask for the device model if it's not specified, as troubleshooting
    steps differ between iOS, Android, and other devices.
    """

    # TODO: Create the SolutionIntegratorAgent
    # - Focus on combining insights into a coherent action plan
    # - Prioritize solutions by likelihood of success and ease of implementation
    # Example system message:
    """
    You are a solution integrator who combines technical analysis into
    actionable plans for customers.
    Your responsibilities:
    1. Synthesize information from the network and device experts
    2. Create a prioritized list of troubleshooting steps
    3. Present solutions in order from simplest to most complex
    4. Estimate which solution is most likely to resolve the issue

    Your final answer should always be a numbered list of actions the customer
    can take, starting with the simplest and most likely to succeed.
    """

    # TODO: Set up the GroupChat
```

```
    # - Include all agents in the group
    # - Configure the chat flow for collaborative problem-solving

    # TODO: Create the GroupChatManager
    # - Set up to manage the conversation flow
    # - Configure to terminate when a solution is reached

    # TODO: Return the user_proxy and manager for interaction
    pass

def process_network_query(query):
    """Process a network troubleshooting query using AutoGen agents"""
    # TODO: Create network agents using create_network_agents function

    # TODO: Initiate the troubleshooting process
    # - Start the conversation with the user's query
    # - Let the agents collaborate to analyze and solve the problem

    # TODO: Extract the final solution
    # - Get the SolutionIntegrator's final response
    # - Format it into a clear troubleshooting plan

    # TODO: Return the formatted solution
    pass
```

1.7.3 LangChain Implementation (Service Recommendations)

[ ]:
```
from langchain.agents import create_react_agent, AgentExecutor
from langchain.tools import SQLDatabaseTool, Tool
from langchain.chat_models import ChatOpenAI
from langchain.prompts import PromptTemplate
from langchain.tools.python.tool import PythonREPLTool

# Define a prompt template for service recommendations
SERVICE_RECOMMENDATION_TEMPLATE = """You are a telecom service advisor who␣
 ↪helps customers find the best plan for their needs.

When recommending plans, consider:
1. The customer's usage patterns (data, voice, SMS)
2. Number of people/devices that will use the plan
3. Special requirements (international calling, streaming, etc.)
4. Budget constraints

Always explain WHY a particular plan is a good fit for their needs.

User query: {query}
```

```
If you need more information, ask for it.
"""

def create_service_agent():
    """Create and return a LangChain agent for service recommendations"""
    # TODO: Create an LLM instance
    # - Initialize ChatOpenAI with model="gpt-3.5-turbo" or "gpt-4"
    # - Set temperature=0.2 for slightly creative but mostly consistent␣
 ↪recommendations

    # TODO: Create tools for the agent
    # - Create a SQLDatabaseTool to query plan information
    #   - Ensure it can access plans, features, pricing, and promotions tables
    # - Add a PythonREPLTool for calculations (e.g., comparing plan costs)
    # - Create a custom Tool for usage estimation based on activity descriptions
    #   Example usage estimation function:
    #   """
    #   def estimate_data_usage(activities):
    #       '''
    #       Estimate monthly data usage based on activities.
    #       Example input: "streaming 2 hours of video daily, browsing 3 hours,␣
 ↪video calls 1 hour weekly"
    #       '''
    #       # Logic to calculate approximate GB needed
    #       return estimated_gb
    #   """

    # TODO: Create the agent prompt
    # - Use the SERVICE_RECOMMENDATION_TEMPLATE
    # - Format it as a PromptTemplate with the appropriate input variables

    # TODO: Create the ReAct agent
    # - Use create_react_agent with the LLM, tools, and prompt
    # - Configure stop sequences and observation format as needed

    # TODO: Create the AgentExecutor
    # - Wrap the agent in an AgentExecutor
    # - Add error handling and max_iterations parameters
    # - Configure verbose=True for development/debugging

    # TODO: Return the agent executor
    pass

def process_recommendation_query(query):
    """Process a service recommendation query using the LangChain agent"""
    # TODO: Create or get the service agent
    # - Call create_service_agent() to build or retrieve the agent
```

```
    # TODO: Process the query
    # - Call agent.run() with the query as input
    # - Handle any errors or agent requests for clarification

    # TODO: Format the response
    # - Structure the recommendation with clear sections
    # - Highlight key benefits and potential savings
    # - Include a summary of why this plan matches their needs

    # TODO: Return the formatted recommendation
    pass
```

1.7.4 LlamaIndex Implementation (Knowledge Retrieval)

[ ]:
```
from llama_index import VectorStoreIndex, SimpleDirectoryReader, ServiceContext
from llama_index.indices.query.schema import QueryMode
from llama_index.query_engine import RouterQueryEngine
from llama_index.selectors.llm_selectors import LLMSingleSelector
from llama_index.llms import OpenAI
from llama_index.tools import QueryEngineTool, SQLDatabase, PandasQueryEngine
import pandas as pd
import os

def create_knowledge_engine():
    """Create and return a LlamaIndex query engine for knowledge retrieval"""
    # TODO: Initialize LLM and service context
    # - Create an OpenAI LLM instance with model="gpt-3.5-turbo" or "gpt-4"
    # - Set temperature=0 for factual responses
    # - Initialize a ServiceContext with the LLM

    # TODO: Load and index documents
    # - Use SimpleDirectoryReader to load documents from data/documents
    # - Process different document types (markdown, PDF, text)
    # - Split documents into chunks of appropriate size
    # - Create embeddings for the document chunks
    # - Build a VectorStoreIndex from the documents

    # TODO: Set up vector search query engine
    # - Configure the vector index query engine
    # - Set similarity_top_k=3 to retrieve most relevant documents
    # - Configure response synthesizer for concise answers

    # TODO: Connect to the database for factual queries
    # - Set up SQLDatabase connection to telecom.db
    # - Create SQL query engine for precise factual lookups
    # - Write prompt that helps translate natural language to SQL
```

```
    # Example SQL generation prompt:
    """
    You are an expert in converting natural language questions about telecom␣
↪services
    into SQL queries. The database contains tables for coverage areas, device␣
↪compatibility,
    and technical specifications.

    When writing SQL:
    1. Use coverage_areas table for location-based questions
    2. Use device_compatibility for phone-specific inquiries
    3. Use technical_specs for network technology questions

    Write focused queries that only retrieve the columns needed to answer the␣
↪question.
    """

    # TODO: Create router query engine
    # - Create QueryEngineTools for both vector and SQL engines
    # - Set up LLMSingleSelector to choose between engines
    # - Create and configure the RouterQueryEngine
    # Example selector prompt:
    """
    You need to determine which query engine to use:

    1. Vector Search Engine: Best for conceptual, procedural questions like␣
↪"How do I set up VoLTE?"
       or "What's the process for international roaming?"

    2. SQL Database Engine: Best for factual, data-driven questions like "Which␣
↪areas have 5G coverage?"
       or "Is the Samsung Galaxy S22 compatible with VoLTE?"

    Based on the query, select the most appropriate engine by analyzing if the␣
↪question
    requires factual data lookup (SQL) or procedural knowledge (Vector).

    Query: {query}
    """

    # TODO: Return the router query engine
    pass

def process_knowledge_query(query):
    """Process a knowledge retrieval query using the LlamaIndex query engine"""
    # TODO: Create or get the knowledge engine
```

```
    # - Call create_knowledge_engine() to build or retrieve the engine

    # TODO: Process the query
    # - Call engine.query() with the query text
    # - Ensure proper error handling

    # TODO: Format the response
    # - Extract the response text
    # - Format with appropriate headings and structure
    # - Add source attribution if available

    # TODO: Return the formatted response
    pass
```

1.8 Main Application Entry Point

[ ]:
```
import streamlit as st
import os
import sys
from pathlib import Path


# Import the Streamlit UI
from ui.streamlit_app import main

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

1.9 Sample Queries for Testing

Use these sample queries to test your implementation:

1.9.1 Billing Queries (CrewAI):

- "Why did my bill increase by 200 this month?"
- "I see a charge for international roaming but I haven't traveled recently"
- "Can you explain the 'Value Added Services' charge on my bill?"
- "What's the early termination fee if I cancel my contract?"

1.9.2 Network Issues (AutoGen):

- "I can't make calls from my home in Mumbai West"
- "My data connection keeps dropping when I'm on the train"
- "Why is my 5G connection slower than my friend's?"
- "I get a 'No Service' error in my basement apartment"

1.9.3 Plan Recommendations (LangChain):

- "What's the best plan for a family of four who watches a lot of videos?"

- "I need a plan with good international calling to the US"
- "Which plan is best for someone who works from home and needs reliable data?"
- "I'm a light user who mostly just calls and texts. What's my cheapest option?"

1.9.4 Technical Information (LlamaIndex):

- "How do I set up VoLTE on my Samsung phone?"
- "What are the APN settings for Android devices?"
- "How can I activate international roaming before traveling?"
- "What areas in Delhi have 5G coverage?"

1.9.5 Edge Cases:

- "Tell me a joke about telecom" (tests fallback handler)
- "I need help with both my bill and network issues" (tests complex query handling)
- " " (empty query tests error handling)

1.10 Conclusion

This project brings together multiple AI frameworks in a practical application for the telecom domain. By successfully implementing this telecom service assistant, you'll demonstrate your ability to:

1. Use LangGraph for orchestrating complex AI workflows
2. Implement CrewAI for specialized collaborative agents
3. Leverage AutoGen for multi-agent conversations
4. Apply LangChain for flexible agent construction
5. Utilize LlamaIndex for effective knowledge retrieval

Good luck, and remember to refer to the course materials when you need guidance on specific implementations!