```
import os
from typing import Any, TypedDict
from langchain_openai import AzureChatOpenAI
from pydantic import BaseModel, Field
from api.app.agents.data_warehouse.agents.data_warehouse_fidelity_wealthscape import
data_warehouse_fidelity_wealthscape_agent
from api.app.config.logger import logger
from api.app.data_sources.redtail.contact.redtail_contact import get_redtail_contact,
get_redtail_contact_notes
from api.app.helpers.azure_postgresql_manager import HouseholdLookupService,
LookupClientAdvisorService, LookupClientNicknameService,
RedtailMeetingPreparationContextService, RedtailNamesLookupService
from datetime import date, datetime, timedelta

from api.app.schemas.redtail_schema import RedtailAccountCredentialsSchema,
RedtailContactNotesSchema, RedtailContactSchema

from langgraph.graph import StateGraph, END, START

from langchain_core.output_parsers import JsonOutputParser

class ApplicabilityToClient(BaseModel):
    client_age: int = Field(..., description="The age of the client.")
    applicable_for_penalty_free_withdrawal: bool = Field(..., description="Whether the client is
applicable for penalty-free withdrawal.")
    applicable_for_social_security_benefits: bool = Field(..., description="Whether the client is
applicable for social security benefits.")
    applicable_for_medicare_benefits: bool = Field(..., description="Whether the client is
applicable for medicare benefits.")
    applicable_for_rmds: bool = Field(..., description="Whether the client is applicable for
RMDs.")
    applicable_for_qcds: bool

class RedtailGraph(TypedDict):
    client_name: str
    last_meeting_date: str

    client_age: str
    client_age_number: int

    client_profile: str
    employment_and_meeting_insights: str
    thresold_evaluation: str
    applicability_to_client: Any
```

```
    latest_meeting_insights: str

    next_steps: str
    timeline_and_milestone_trigger: str

    all_past_meetings: str
    latest_agenda_info: str
    next_agenda_topics: str
    latest_client_info: str

    external_account_info: str

def get_client_age(
    input: RedtailGraph
):
    """
    Gets the client age for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        logger.info(f"Client Name: {client_name}")

        client_age = data_warehouse_fidelity_wealthscape_agent(
            user_full_requirement=f""" How old is {client_name}? Return exact numbers, NO
DECIMALS.""",
            is_ui_based=True
        )

        logger.info(f"Client Age: {client_age}")

        client_age_number = int(round(float(model.invoke(f"""
```

```
                                    Given this message, {client_age}, only return the age as a number. If
2 or more ages are provided, return the oldest age. No other text is required.
                                    """).content)))

        logger.info(f"Client Age Number: {client_age_number}")

        return {'client_age': client_age, 'client_age_number': client_age_number}

    except Exception as e:
        logger.error(f"Error in get_client_age: {e}")
        raise

def build_client_profile(
    input: RedtailGraph
):
    """
    Builds a client profile for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        logger.info(f"Client Name: {client_name}")

        orm_instance_advisor = LookupClientAdvisorService()

        advisor_name = orm_instance_advisor.get_latest_record(
            client_name=client_name
        )['advisor_name']

        nickname_instance = LookupClientNicknameService()

        nickname_record = nickname_instance.get_latest_record(
            client_name=client_name
        )
```

```
    nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname' in nickname_record else None

    orm_instance = RedtailMeetingPreparationContextService()

    redtail_context = orm_instance.get_context_by_client(
        client_name=client_name,
        advisor_name=advisor_name
    )

    today = date.today().strftime("%B %d, %Y")

    prompt = f"""
    You are a senior AI assistant for a financial planning team.

    Today's date is **{today}**. You have been given **Redtail CRM data** that spans across different points in time — including emails, advisor notes, Redtail Speak logs, or CRM fields. This data evolves temporally: older notes may contain outdated or overwritten information. Your role is to extract a current and clean **Client Profile** using only what is explicitly supported in this data.

    THIS IS THE CLIENT AGE: {input['client_age']}. DON'T USE ANY OTHER AGE.

    ⚠️ Treat the context as **temporal**, not static. If multiple values exist for a field, prefer the most recent entry based on timestamps or language cues. If something is ambiguous or not fully clear, flag it for human review — do **not guess**.

    ---

    ### Output: Structured Client Profile with Chain of Thought (CoT) Justification

    Use the following format. For **each field**, include a reasoning block that explains:

    1. **Where it was found**
    2. **Why it was selected (especially if multiple/conflicting values exist)**
    3. **Any uncertainty or missing data flag**

    ---

    #### 🗒️ Demographics
    - **Full Legal Name**:
    > Reasoning:
    - **Gender**:
```

> Reasoning:

👨‍👩‍👧‍👦 Family & Relationship Context
- **Marital Status**:
> Reasoning:
- **Spouse/Partner Name**:
> Reasoning:
- **Spouse/Partner DOB**:
> Reasoning:
- **Children / Grandchildren**:
> Reasoning:
- **POAs (Power of Attorney)**:
> Reasoning:
- **Death Dates (Spouse / Parent / IRA Originator)**:
> Reasoning:

🏦 Account Ownership Context
- **Account Ownership Type (e.g., individual, joint, trust)**:
> Reasoning:
- **Trust-held or Named Trust Entities**:
> Reasoning:
- **Inherited IRA Status (if stated)**:
> Reasoning:

⚠️ Special Conditions or Flags
- **Disability / Terminal Illness / LTC Status**:
> Reasoning:
- **Veteran or Military Status**:
> Reasoning:
- **Non-citizen or Immigration Flag**:
> Reasoning:
- **Legal Custodianship / Special Needs**:
> Reasoning:

🧠 Client Philosophy & Behavioral Markers
- **Legacy or Inheritance Goals**:

```
        > Reasoning:
        - **Risk Tolerance / Investment Beliefs**:
        > Reasoning:
        - **Charitable Giving Intentions**:
        > Reasoning:
        - **Ethical or ESG Constraints**:
        > Reasoning:

        ---

        ⚠️ **If a detail is not available**, respond with:
        `Not indicated`
        And in reasoning, say:
        > "This information is not present or inferable from the provided context."

        ⚠️ **Do not infer or assume. All values must be directly supported by the context.**

        ---

        Client's Full Name: {client_name}
        Client's Advisor: {advisor_name}
        Client's Nickname (if applicable): {nickname}

        ---

        ### Input: Redtail CRM Context (Unstructured)
        {redtail_context['content']}
        """

        response = model.invoke(prompt).content

        logger.info(f"Client Profile: {response}")

        return {'client_profile': response}
    except Exception as e:
        logger.error(f"Error in build_client_profile: {e}")
        raise

def build_employment_and_meeting_insights(
    input: RedtailGraph
):
    """
    Builds employment and meeting insights for a given client name.
    """
```

```
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        logger.info(f"Client Name: {client_name}")

        orm_instance_advisor = LookupClientAdvisorService()

        advisor_name = orm_instance_advisor.get_latest_record(
            client_name=client_name
        )['advisor_name']

        nickname_instance = LookupClientNicknameService()

        nickname_record = nickname_instance.get_latest_record(
            client_name=client_name
        )

        nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

        orm_instance = RedtailMeetingPreparationContextService()

        redtail_context = orm_instance.get_context_by_client(
            client_name=client_name,
            advisor_name=advisor_name
        )

        today = date.today().strftime("%B %d, %Y")

        prompt = f"""
        You are a senior AI assistant for a financial planning team.
        Today's date is **{today}**.

        THIS IS THE CLIENT AGE: {input['client_age']}. DON'T USE ANY OTHER AGE.
```

You have received temporal Redtail CRM data (emails, notes, Redtail Speak logs, summaries), which may include references to employment, meeting cadence, and client engagement. This data may evolve across time, so treat the context as **non-static** — newer updates may override older ones.

🎯 Objective

Extract two key groups of information:

1. **Employment & Retirement Status**
2. **Meeting Frequency, Cadence, and Review Labels**

For each data point, use **Chain of Thought reasoning** to explain:
- Where the data came from
- Why it was chosen
- Whether it is current or outdated
- Any conflicts or missing context

🔍 Employment & Retirement Data
- **Employment Status (Current)**:
> Reasoning:
- **Retirement Intentions / Timeline**:
> Reasoning:
- **Employer Plan Participation (401k, pension, benefits)**:
> Reasoning:
- **Medicare Deferral or Trigger Context**:
> Reasoning:
- **Any Mention of Exit Planning or Buyouts**:
> Reasoning:
- **Work Status Impacting Other Areas (e.g., tax, savings, insurance)**:
> Reasoning:

📅 Meeting & Engagement Profile
- **Last Documented Meeting or Review Date**:
> Reasoning:
- **Meeting Frequency / Cadence (e.g., quarterly, annual)**:
> Reasoning:

- **Client Meeting Style or Review Type** (A, B, C, D Reviews):
> Reasoning:
- **Topics Typically Covered in Their Reviews**:
> Reasoning:
- **Engagement Style** (e.g., responsive, passive, skips follow-ups):
> Reasoning:
- **Any Missed, Postponed, or Skipped Reviews**:
> Reasoning:
- **Notes on Preferences or Patterns**:
> Reasoning:

🧠 Matching Logic

- Reviews may be labeled as: **Review A**, **Review B**, **Review C**, or **Review D**. All of these are considered structured meetings.
- Clients may say “check-in”, “annual review”, or “meeting” — normalize all of these under the “Review” concept.
- Use latest timestamp or language (e.g., “as of now”, “still working”, “retiring this fall”) to choose the **most current version** of employment data.
- If a meeting is skipped or canceled, still count the intended cadence (e.g., “client usually meets quarterly but skipped Q2”).
- If client is age 65+ and marked as “still working,” flag **Medicare deferral** planning as potentially relevant.

⚠️ **If any detail is not explicitly available**, return `Not indicated` and explain:
> "This detail was not present or inferable from the provided Redtail context."

⚠️ **Do not infer or guess.** Rely strictly on Redtail input, but use professional judgment to interpret updates over time.

Client Full Name: {client_name}
Client Advisor: {advisor_name}
Client Nickname (if applicable): {nickname}

Redtail CRM Input (Unstructured)
{redtail_context['content']}

```
        """

        response = model.invoke(prompt).content

        logger.info(f"Employment and Meeting Insights: {response}")

        return {'employment_and_meeting_insights': response}
    except Exception as e:
        logger.error(f"Error in build_employment_and_meeting_insights: {e}")
        raise

def build_thresold_evaluation_for_client(
    input: RedtailGraph
):
    """
    Builds a threshold evaluation for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        logger.info(f"Client Name: {client_name}")

        orm_instance_advisor = LookupClientAdvisorService()

        advisor_name = orm_instance_advisor.get_latest_record(
            client_name=client_name
        )['advisor_name']

        nickname_instance = LookupClientNicknameService()

        nickname_record = nickname_instance.get_latest_record(
            client_name=client_name
        )
```

```
nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname' in nickname_record else None

orm_instance = RedtailMeetingPreparationContextService()

redtail_context = orm_instance.get_context_by_client(
    client_name=client_name,
    advisor_name=advisor_name
)

today = date.today().strftime("%B %d, %Y")

prompt = f"""
You are a highly specialized AI working alongside financial advisors.
Today's date is **{today}**.

THIS IS THE CLIENT AGE: {input['client_age']}. DON'T USE ANY OTHER AGE.

You've been given a historical record of Redtail CRM data (including emails, notes, Redtail Speak logs, and summaries). These entries span **multiple time periods** and may include outdated, replaced, or evolving information. Treat the timeline as **non-linear and dynamic**.

Your job is to extract a **structured insight profile** about the client — drawing only from information in the Redtail data, and using professional, temporal, and logical judgment.

---

### 🧠 Core Rules You Must Follow

1. **Meetings = Reviews**:
Normalize all phrases like "meeting", "check-in", "touch base", "Review A/B/C/D", or "Q2 review" under the concept of "Review".

2. **Time Evolution**:
If conflicting data exists (e.g., working vs. retired), choose the **most recent**. If timing is unclear, **flag for advisor**.

3. **Chain of Thought (CoT) Required**:
For every field, include a reasoning block that:
- States where the data came from
- Explains why it was selected
- Flags any ambiguity or temporal conflict

4. **NEVER guess**. Flag anything uncertain with `Advisor Review Required`.
```

📋 OUTPUT FORMAT — CLIENT INSIGHT PROFILE

---

1. 🧑‍💼 Client-Directed Topics & Personal Concerns
- **Client-Stated Requests or Worries**:
> Reasoning:
- **Major Life Events (past or upcoming)**:
> Reasoning:
- **Self-described Goals or Priorities** (e.g., retiring early, buying property):
> Reasoning:

---

2. 📅 Time-Sensitive Milestones
⚠️ Handle with extreme care.
Use any explicit **birthdate**, or infer from phrases like “turning 65 next year.”

Check if the client is near or past the following milestone **trigger ages**:
- **59½** – Penalty-free retirement withdrawals
- **62** – Social Security eligibility
- **65** – Medicare enrollment window
- **70+** – Qualified Charitable Distributions (QCDs)
- **73** – Required Minimum Distributions (RMDs)

📌 **Required Minimum Distribution (RMD) Logic**:
- **First RMD**: Can be delayed until **April 1 of the year following** the year the client turns 73
- **Subsequent RMDs**: Must be taken by **December 31** of each applicable year
- If first RMD is delayed, client may have to take **two RMDs in the same calendar year**
- Must confirm **IRA type**, **ownership**, and **inheritance status** if unclear

Extracted Milestone Data:
> Reasoning:
- **59½ Trigger**:
> Reasoning:
- **62 – Social Security**:
> Reasoning:
- **65 – Medicare Planning**:
> Reasoning:
- **70+ – QCD Consideration**:

> Reasoning:
- **73 – RMD Planning**:
> Reasoning:
- **Any Phrases Like "Turning 65 next year"**:
> Reasoning:

---

3. 📅 Review History & Engagement Style
Normalize all review formats (e.g., “Review A”, “Q3 check-in”, “Annual meeting”) into this structure:

- **Most Recent Review** (include month/year if available):
> Reasoning:
- **Preferred Cadence** (e.g., quarterly, semi-annual):
> Reasoning:
- **Typical Review Labels Used** (A/B/C/D or others):
> Reasoning:
- **Missed or Skipped Reviews** (if noted):
> Reasoning:
- **Topics Usually Covered in Reviews**:
> Reasoning:
- **Engagement Style** (e.g., proactive, passive, avoids meetings):
> Reasoning:

---

4. 👨‍👩‍👧‍👦 Family & Legal Relationships
- **Spouse or Partner Info** (name, age/DOB if present):
> Reasoning:
- **Children / Grandchildren**:
> Reasoning:
- **Death Dates** (spouse, parent, IRA originator):
> Reasoning:
- **POAs / Executors / Trustees**:
> Reasoning:

---

5. 🧠 Behavioral & Planning Philosophy
- **Risk Tolerance** (explicit or inferred):
> Reasoning:
- **Legacy / Gifting / Inheritance Goals**:
> Reasoning:

- **Charitable Preferences or Faith-based Directives**:
> Reasoning:
- **Communication or Decision-Making Style**:
> Reasoning:

---

6. 🏦 Account & Ownership Context
- **Account Titles or Ownership Types (trust, joint, individual)**:
> Reasoning:
- **Mention of Trusts / Successor Trustees**:
> Reasoning:
- **Inherited IRA Context**:
> Reasoning:
- **Uninvested Cash / Untouched Accounts**:
> Reasoning:

---

7. ✅ Action Items, Pending Loops & Follow-ups
- **Outstanding Tasks or Promised Follow-ups**:
> Reasoning:
- **Unresolved Topics from Past Reviews**:
> Reasoning:
- **Advisor Commitments Still Open**:
> Reasoning:
- **Forms/Documents/Applications Awaiting Completion**:
> Reasoning:

⚠️ Final Rule:
Do NOT hallucinate. All extracted information must be grounded in the Redtail input. If information is missing or unclear, respond with:
`Not indicated`
And explain:
> "No reference found or date unclear; flagged for advisor review."

Client Name: {client_name}
Advisor: {advisor_name}
Nickname (if applicable): {nickname}

```
        ---

        ### 🔍 Redtail Context (Temporal + Unstructured)
        {redtail_context['content']}
        """

        response = model.invoke(prompt).content

        logger.info(f"Threshold Evaluation for Client: {response}")

        json_parser = JsonOutputParser(pydantic_object=ApplicabilityToClient)

        prompt_for_applicability_to_client = f"""
        You are a financial planning assistant using Redtail CRM data to support client agenda preparation.
        Today’s date is **{today}**.

        THIS IS THE CLIENT AGE: {input['client_age']}. DON'T USE ANY OTHER AGE.

        You have been provided with a threshold evaluation for a given client.
        Your job is to extract a structured insight profile about the client — drawing only from information in the Redtail data, and using professional, temporal, and logical judgment.

        Check if the client is near or past the following milestone **trigger ages**:
        - **59½** – Penalty-free retirement withdrawals
        - **62** – Social Security eligibility
        - **65** – Medicare enrollment window
        - **70+** – Qualified Charitable Distributions (QCDs)
        - **73** – Required Minimum Distributions (RMDs)

        Context: {response}

        ### You must format your response in the following JSON format:
        {json_parser.get_format_instructions()}
        """

        applicability_to_client = json_parser.invoke(model.invoke(prompt_for_applicability_to_client).content)

        logger.info(f"Applicability to Client: {applicability_to_client}")

        return {'thresold_evaluation': response, 'applicability_to_client': applicability_to_client}
    except Exception as e:
        logger.error(f"Error in build_thresold_evaluation_for_client: {e}")
```

```
        raise

def extract_latest_meeting_insights(
    input: RedtailGraph
):
    """
    Extracts the latest meeting insights for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        logger.info(f"Client Name: {client_name}")

        orm_instance_advisor = LookupClientAdvisorService()

        advisor_name = orm_instance_advisor.get_latest_record(
            client_name=client_name
        )['advisor_name']

        nickname_instance = LookupClientNicknameService()

        nickname_record = nickname_instance.get_latest_record(
            client_name=client_name
        )

        nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

        orm_instance = RedtailMeetingPreparationContextService()

        redtail_context = orm_instance.get_context_by_client(
            client_name=client_name,
            advisor_name=advisor_name
        )
```

```
today = date.today().strftime("%B %d, %Y")

redtail_names_lookup_service = RedtailNamesLookupService()

result = redtail_names_lookup_service.get_latest_record(
    name=client_name
)

client_result = get_redtail_contact(
    input = RedtailContactSchema(
        **{
            "first_name": result['first_name'],
            "last_name": result['last_name']
        }
    ),
    account = RedtailAccountCredentialsSchema(
        **{
            "username": os.environ.get("RT_USERNAME"),
            "password": os.environ.get("RT_PASSWORD")
        }
    )
)

print(client_result)

notes = get_redtail_contact_notes(
    input = RedtailContactNotesSchema(
        **{
            "contact_id": str(client_result['id']),
            "start_date": input['last_meeting_date'],
            "end_date": datetime.now().strftime("%Y-%m-%d")
        }
    ),
    account = RedtailAccountCredentialsSchema(
        **{
            "username": os.environ.get("RT_USERNAME"),
            "password": os.environ.get("RT_PASSWORD")
        }
    )
)

prompt = f"""
```

You are a senior financial planning assistant using Redtail CRM data to support client agenda preparation.
Today's date is **{today}**.

THIS IS THE CLIENT AGE: {input['client_age']}. DON'T USE ANY OTHER AGE.

You have been provided with historical Redtail notes, Redtail Speak logs, and activity entries related to a specific client. Your job is to extract structured insight from the **latest client review** (also called meeting, check-in, or Review A/B/C/D).

⚠️ Key Instructions

1. Normalize all terms like "meeting", "check-in", "Review A", "Review C", etc. — treat all of them as **reviews**.
2. Identify the **most recent review** using dates or time markers. Ignore earlier ones.
3. Use **Chain of Thought (CoT) reasoning** for every item extracted:
- Where the data came from
- Why it was selected
- Whether it is ambiguous or requires advisor review
4. Include **client-originated content** (questions, concerns, updates, or requests).
5. If no recent review is clearly present, respond: `Latest review not clearly indicated — Advisor Review Required`.

📋 OUTPUT: Latest Client Review Summary (with Reasoning)

---

1. 🗒️ Topics Discussed or Resolved
- **Planning topics brought up** (by client or advisor):
> Reasoning:
- **Decisions made during this review**:
> Reasoning:

---

2. ❓ Client Questions, Concerns, or Requests
- **Client-originated concerns** (e.g., market anxiety, inflation fears):
> Reasoning:
- **Direct client requests or asks** (e.g., "Can we update my beneficiaries?"):
> Reasoning:

- **Questions the client asked**:
> Reasoning:
- **Client updates voluntarily shared** (life, health, job, etc.):
> Reasoning:

---

3. ⏭️ Deferred or Incomplete Topics
- **Topics postponed to next review**:
> Reasoning:
- **Advisor follow-ups still outstanding**:
> Reasoning:

---

4. 🔁 New Client Info Since Last Review
- **New family or financial developments**:
> Reasoning:
- **Shift in client goals or timelines**:
> Reasoning:

---

5. ✅ Action Items & Next Steps
- **Advisor responsibilities following the review**:
> Reasoning:
- **Client responsibilities or tasks**:
> Reasoning:
- **Event-based or time-sensitive triggers**:
> Reasoning:

---

6. 🧠 Behavioral or Emotional Insights
- **Client reactions to plans or performance**:
> Reasoning:
- **Risk tolerance or communication changes**:
> Reasoning:

---

7. 📅 Review Preferences Going Forward
- **Stated cadence or review frequency updates**:
> Reasoning:

```
        - **Format or delivery preferences (Zoom, portal, email, in-person)**:
        > Reasoning:

        ---

        Client Name: {client_name}
        Advisor: {advisor_name}
        Nickname (if applicable): {nickname}


        Previos Context to take into account:
        Client Profile: {input['client_profile']}
        Employment and Meeting Insights: {input['employment_and_meeting_insights']}
        Threshold Evaluation: {input['thresold_evaluation']}

        ---

        ### 🔍 Input: Redtail CRM History (multi-date, unstructured)
        {notes}
        """

        response = model.invoke(prompt).content

        logger.info(f"Latest Meeting Insights for Client: {response}")

        return {'latest_meeting_insights': response}
    except Exception as e:
        logger.error(f"Error in extract_latest_meeting_insights: {e}")
        raise

def extract_next_steps(
    input: RedtailGraph
):
    """
    Extracts the next steps for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
```

```
    )

    household_service = HouseholdLookupService()
    client_name = household_service.find_household_by_individual(input['client_name'])

    logger.info(f"Client Name: {client_name}")

    orm_instance_advisor = LookupClientAdvisorService()

    advisor_name = orm_instance_advisor.get_latest_record(
        client_name=client_name
    )['advisor_name']

    nickname_instance = LookupClientNicknameService()

    nickname_record = nickname_instance.get_latest_record(
        client_name=client_name
    )

    nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

    orm_instance = RedtailMeetingPreparationContextService()

    redtail_context = orm_instance.get_context_by_client(
        client_name=client_name,
        advisor_name=advisor_name
    )

    today = date.today().strftime("%B %d, %Y")

    redtail_names_lookup_service = RedtailNamesLookupService()

    result = redtail_names_lookup_service.get_latest_record(
        name=client_name
    )

    client_result = get_redtail_contact(
        input = RedtailContactSchema(
            **{
                "first_name": result['first_name'],
                "last_name": result['last_name']
            }
        ),
```

```
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
            }
        )
    )

    notes = get_redtail_contact_notes(
        input = RedtailContactNotesSchema(
            **{
                "contact_id": str(client_result['id']),
                "start_date": input['last_meeting_date'],
                "end_date": datetime.now().strftime("%Y-%m-%d")
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
            }
        )
    )

    prompt = f"""
    You are a senior AI assistant supporting financial advisors with Redtail CRM analysis.
    Today's date is **{today}**.

    THIS IS THE CLIENT AGE: {input['client_age']}. DON'T USE ANY OTHER AGE.

    You are given a full historical Redtail context (including notes, emails, activities, and Redtail Speak logs) related to a specific client. This data may span months or years.

    Your goal is to **generate a list of Advisor Next Steps** based on data that occurred **after the latest review** (where "review" = Review A/B/C/D, check-in, annual meeting, etc.).

    ---

    ### 🧠 Logic You Must Apply

    1. Identify the **latest review** using timestamps or review labels. Ignore older meetings.
    2. Search all Redtail entries **after** that review date to identify client outreach, incomplete workflows, or advisor actions.
    3. Use this logic for deciding what becomes a Next Step:
```

- If a **client reached out** after the last review with a request, question, or concern → **generate a follow-up step**.
- If an **action was completed** but not confirmed with the client → **include it as a recap step**.
- If an activity is **incomplete or partially complete** → **flag it as a check-in item**.
- If an item was discussed in a previous review but not mentioned again → **assume it's resolved**, unless a note contradicts that.

4. For each task, provide a short **justification**:
- Where it was found in the data
- Why it's still relevant or flagged
- Whether confirmation is needed

📋 OUTPUT FORMAT — Advisor Next Steps

Each next step should be clearly actionable and written as a bullet, followed by a **"Reasoning:"** block.

🔁 Advisor Next Steps After Latest Review

• [Next Step 1]
> Reasoning:
> - Based on: [Redtail note, activity, or client message]
> - Why it matters: [e.g., client asked but no advisor follow-up is shown]
> - Confirmation: [Flag if still pending or partially addressed]

• [Next Step 2]
> Reasoning:
> - Based on: ...
> - Why it matters: ...
> - Confirmation: ...

(Repeat for all valid next steps found in the data)

⚠️ CAUTION:
- Do NOT fabricate. Use only what's available in the context.
- If uncertain whether a task is resolved, write: `Flag for advisor confirmation`
- If no next steps are clearly outstanding, respond with: `No next steps identified after the latest review — Advisor may confirm`

```
        ---

        Client Name: {client_name}
        Advisor: {advisor_name}
        Nickname (if applicable): {nickname}

        Previous Context to take into account:
        Client Profile: {input['client_profile']}
        Employment and Meeting Insights: {input['employment_and_meeting_insights']}
        Threshold Evaluation: {input['thresold_evaluation']}
        Latest Meeting Insights: {input['latest_meeting_insights']}
        ---

        ### 🗂️ Redtail Context (Multi-date CRM + Activity Stream)
        {redtail_context['content']}
        """

        response = model.invoke(prompt).content

        logger.info(f"Next Steps for Client: {response}")

        return {'next_steps': response}
    except Exception as e:
        logger.error(f"Error in extract_next_steps: {e}")
        raise

def timeline_and_milestone_trigger(
    input: RedtailGraph
):
    """
    Extracts the timeline and milestone trigger for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
```

```
    client_name = household_service.find_household_by_individual(input['client_name'])

    logger.info(f"Client Name: {client_name}")

    orm_instance_advisor = LookupClientAdvisorService()

    advisor_name = orm_instance_advisor.get_latest_record(
        client_name=client_name
    )['advisor_name']

    nickname_instance = LookupClientNicknameService()

    nickname_record = nickname_instance.get_latest_record(
        client_name=client_name
    )

    nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

    orm_instance = RedtailMeetingPreparationContextService()

    redtail_context = orm_instance.get_context_by_client(
        client_name=client_name,
        advisor_name=advisor_name
    )

    today = date.today().strftime("%B %d, %Y")

    redtail_names_lookup_service = RedtailNamesLookupService()

    result = redtail_names_lookup_service.get_latest_record(
        name=client_name
    )

    client_result = get_redtail_contact(
        input = RedtailContactSchema(
            **{
                "first_name": result['first_name'],
                "last_name": result['last_name']
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
```

```
            "password": os.environ.get("RT_PASSWORD")
        }
    )
)

notes = get_redtail_contact_notes(
    input = RedtailContactNotesSchema(
        **{
            "contact_id": str(client_result['id']),
            "start_date": input['last_meeting_date'],
            "end_date": datetime.now().strftime("%Y-%m-%d")
        }
    ),
    account = RedtailAccountCredentialsSchema(
        **{
            "username": os.environ.get("RT_USERNAME"),
            "password": os.environ.get("RT_PASSWORD")
        }
    )
)

prompt = f"""
You are a financial intelligence assistant trained to analyze Redtail CRM data.
Your goal is to extract a structured summary of all **time-sensitive planning triggers and milestone dates** relevant to the client.

Today’s date is **{today}**.

THIS IS THE CLIENT AGE: {input['client_age']}. DON'T USE ANY OTHER AGE.

You’ve received a historical Redtail context with emails, notes, activities, and review entries. Treat the data as evolving — later entries may supersede earlier ones. Normalize all mentions of "meetings" as "reviews".

---

### 🎯 OBJECTIVE: Timeline & Milestone Trigger Extraction

Client Age: {input['client_age']}

You must:
1. Identify upcoming or relevant **milestone ages or dates** (see below).
2. Apply planning logic — **don’t rely solely on age**:
```

- Consider **employment status**, **disability**, **survivor benefits**, **dependents**, **terminal illness**, etc.
- Adjust Social Security, Medicare, RMDs, or QCD logic accordingly.
3. Assess **time since last review**.
4. Include **Chain of Thought (CoT)** justification for every trigger flagged.
5. If no trigger is present or information is missing, flag for `Advisor Review`.

📅 AGE-BASED MILESTONES TO DETECT:

- **Turning 59½**: Eligible for penalty-free IRA/401(k) withdrawals
- **Turning 62**: Social Security eligibility
- **Turning 65**: Medicare enrollment window (3 months before birth month)
- **Age 70+**: QCD eligibility from IRAs
- **Turning 73**: RMD trigger under SECURE 2.0
- If first RMD year: can delay until **April 1** of following year
- All subsequent RMDs due by **Dec 31 each year**

👩‍💼 EMPLOYMENT CONTEXT CHECKPOINTS:

- **Still working past 65**? → May delay Medicare Part B
- **Approaching retirement**? → Consider triggering planning tools (RightCapital, RetireUp)
- **Recent job loss or job change** → May impact HSA, 401(k), Roth strategy
- **Disability or terminal illness** → May qualify for special distribution exemptions

---

🧠 OUTPUT FORMAT – TIMELINE & TRIGGERS (with Reasoning)

⏱️ Time Since Last Review
- **Last Review Date**:
> Reasoning:
- **Time elapsed**:
> Reasoning:

🧾 Detected Milestones and Eligibility Triggers

• **[Trigger Description]** (e.g., "Client turning 65 in March 2026 – Medicare enrollment required by Dec 2025")
> Reasoning:
> - Found in: [Redtail note, activity, or Speak log]
> - Planning relevance: [Explain importance of trigger]
> - Adjustments: [e.g., client still working, delaying Medicare B]
> - Confirmation status: [e.g., confirmed in note, still unclear, or flag for review]

(Repeat for every milestone or inferred trigger)

⚠️ If No Triggers Are Found
If no age-based or context-based milestones are present, return:
`No relevant planning triggers identified based on current Redtail data. Advisor should review for silent assumptions.`

Client Name: {client_name}
Advisor: {advisor_name}
Nickname (if applicable): {nickname}

Previous Context to take into account:
Client Profile: {input['client_profile']}
Employment and Meeting Insights: {input['employment_and_meeting_insights']}
Threshold Evaluation: {input['thresold_evaluation']}
Latest Meeting Insights: {input['latest_meeting_insights']}

🔍 Redtail CRM Context (Evolving Timeline)
{notes}
"""

response = model.invoke(prompt).content

logger.info(f"Timeline and Milestone Trigger for Client: {response}")

return {'timeline_and_milestone_trigger': response}
except Exception as e:
logger.error(f"Error in timeline_and_milestone_trigger: {e}")
raise

```
def extract_all_past_meetings(
    input: RedtailGraph
):
    """
    Extracts all past meetings for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        orm_instance_advisor = LookupClientAdvisorService()

        advisor_name = orm_instance_advisor.get_latest_record(
            client_name=client_name
        )['advisor_name']

        nickname_instance = LookupClientNicknameService()

        nickname_record = nickname_instance.get_latest_record(
            client_name=client_name
        )

        nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

        orm_instance = RedtailMeetingPreparationContextService()

        redtail_context = orm_instance.get_context_by_client(
            client_name=client_name,
            advisor_name=advisor_name
        )

        redtail_context_raw = redtail_context['content']
```

prompt = f"""
You are an analyst who consolidates scattered advisor notes, workflows, and emails into a clean, chronological meeting history **without adding conclusions unless explicitly requested**. Extract only what is documented and present it in a clear, structured report with a compact timeline table.

Audience & Tone
- Audience: financial advisors, compliance officers, and operations staff.
- Output language: **English**.
- Style: factual, neutral, precise.
- No speculation, no inferred advice, no extra commentary.

Inputs
- Client Name: {client_name}
- Advisor Name: {advisor_name}
- Nickname (if applicable): {nickname}
- Redtail Context: {redtail_context_raw}

Coverage & Parsing Rules (must follow)
- Ingest **all** provided note batches end-to-end (batch 1..N). No skipping or early stopping.
- Parse dates from multiple formats:
- ISO: YYYY-MM-DD (e.g., 2025-09-11)
- Month Day, Year (e.g., August 5, 2025)
- Day-Month-Year (e.g., 05-Feb-2025)
- **Ranges with en dash**: “July 18–23, 2024”. Use the **start date** (18-Jul-2024) for chronology and retain the full range text in the event title.
- Normalize all dates to **DD-MMM-YYYY** in English (e.g., 11-Sep-2025). Group events by **year** and sort strictly ascending within each year.
- **Must include every event through the latest date present in the input notes** (do not stop at earlier months). If later items exist (e.g., August–September), they must appear.
- If a record has a date and an action but is not labeled as a “meeting”, include it as an engagement (e.g., workflows, activities, trades, distributions, compliance emails, estate docs).

De-duplication & Merge
- Merge duplicates across batches; prefer the most detailed record.
- Preserve exact staff names and completion dates in **Outcome/Validation**.

Recency-First Retention (when length constraints apply)
- If compression is needed due to length, **preserve the most recent quarter in full detail first** (e.g., Aug–Sep 2025) and then compress older periods. Do **not** drop any events from the newest quarter.

Extraction Rules
1. **No conclusions**: Do not add a “Conclusion” section unless explicitly requested.
2. **No hallucinations**: If the notes do not specify something, write “(no detail in notes)”.
3. **Validation hooks**: Always include workflow completions, emails, or staff names under **Outcome/Validation**.
4. **Names & roles**: Preserve exactly as written.
5. **Currency**: Keep as-is (e.g., $10,000). Do not convert.
6. **Acronyms**: Keep as-is (RMD, QCD, ADV 2A, etc.).
7. **Bullets**: Use **expanded bullets** for topics (aim 5–7 per event when possible).
8. **Gaps**: If any field is missing, write “(no detail in notes)”.

Required Output Structure
1. **Title**: “Consolidated Commentary of Past Meetings — {client_name}”
2. **Sections by Year**
- For **each** meeting/engagement:
  - **Date — Type**
  - **Topics**: multiple detailed bullets (expanded).
  - **Outcome/Validation**: staff responsible, workflows/activities/emails with completion dates.
3. **Timeline Table (Operational Summary)**
- Columns (exact): Date | Type | Channel/Location | Activity/Title | Key Topics | Outcome / Validation | Responsible
- One row per event; concise but informative cells.
4. **Registry Notes (Internal Use)**
- 4–8 operational reminder bullets.
- Strictly operational. No conclusions.

Formatting Rules
- Use Markdown headings (#, ##, ###) and bold labels.
- Months abbreviated in English: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec.
- Keep table formatting in Markdown.
- Do **not** include a “Conclusion” section unless explicitly requested.
"""

result = model.invoke(prompt).content

```
        logger.info(f"All Past Meetings for Client: {result}")

        return {'all_past_meetings': result}

    except Exception as e:
        logger.error(f"Error in extract_all_past_meetings: {e}")
        raise

def retrieve_latest_agenda_info(
    input: RedtailGraph
):
    """
    Retrieves the latest agenda info for a given client name.
    """
    try:

        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        orm_instance_advisor = LookupClientAdvisorService()

        advisor_name = orm_instance_advisor.get_latest_record(
            client_name=client_name
        )['advisor_name']

        nickname_instance = LookupClientNicknameService()

        nickname_record = nickname_instance.get_latest_record(
            client_name=client_name
        )

        nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

        today = date.today().strftime("%B %d, %Y")
```

```
    redtail_names_lookup_service = RedtailNamesLookupService()

    result = redtail_names_lookup_service.get_latest_record(
        name=client_name
    )

    client_result = get_redtail_contact(
        input = RedtailContactSchema(
            **{
                "first_name": result['first_name'],
                "last_name": result['last_name']
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
            }
        )
    )

    notes = get_redtail_contact_notes(
        input = RedtailContactNotesSchema(
            **{
                "contact_id": str(client_result['id']),
                "start_date": input['last_meeting_date'],
                "end_date": datetime.now().strftime("%Y-%m-%d")
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
            }
        )
    )

    print("Latest Meeting Date: ", (datetime.strptime(input['last_meeting_date'], "%Y-%m-%d")
+ timedelta(days=1)).strftime("%Y-%m-%d"))

    prompt = f"""
    You are an analyst who extracts the full agenda and discussion topics of the most recent
client meeting.
```

You are given:
- a **Reference Date** (today/current date),
- a **Target Meeting Date** (which may be exact or approximate),
- and a corpus of meeting notes/activities.

Your job is to select the latest **Review** meeting (Office Review, In-Person Review, Mobile Review, etc.; may be labeled Review A/B/C/D) that:
1) occurs **on or before** the Reference Date, and
2) falls **within a ±2-day window** around the Target Meeting Date (when provided or implied).

Then return **only**:
- **Agenda (Content)** (verbatim/faithful to the notes; preserve order and wording),
- **Topics Discussed** (expanded bullets; factual, no speculation).

Audience & Tone
- Audience: financial advisors, compliance officers, operations staff.
- Output language: **English**.
- Style: factual, neutral, precise.
- No speculation, no inferred advice, no extra commentary.

Inputs
- Client Name: {client_name}
- Reference Date: {today}
- Target Meeting Date: {(datetime.strptime(input['last_meeting_date'], "%Y-%m-%d") + timedelta(days=1)).strftime("%Y-%m-%d")}
- Redtail Context: {notes}

Selection & Date Handling Rules
- Ingest all meetings/engagements in the context end-to-end.
- Parse dates from multiple formats:
- ISO: YYYY-MM-DD (e.g., 2025-09-11)
- Month Day, Year (e.g., August 20, 2025)
- Day-Month-Year (e.g., 20-Aug-2025, 05-Feb-2025)
- Ranges with en dash (e.g., July 18–23, 2024). Treat the **start date** as the anchor for chronology and retain the range text in the meeting type/title when relevant.
- Define a **tolerance window**: Target Meeting Date ± 2 days.
- Prefer meetings whose Meeting Date (or range) intersects this window.
- If **multiple** meetings intersect, choose the one **closest to** the Target Meeting Date; if still tied, choose the **latest by date** ≤ Reference Date.
- If **no meeting** intersects the ±2-day window, choose the **latest Review meeting ≤ Reference Date** (fallback).

- Meetings are **Reviews** (Office/In-Person/Mobile/etc.) and may be marked **Review A/B/C/D**. Non-Review items (e.g., workflows/emails only) are **not** valid unless explicitly labeled as a Review.

🔒 STRICT EXTRACTION RULES

You must return **only two sections** for the selected meeting:

1) **Agenda (Content)**

✅ **What qualifies as an Agenda**:
- A document that explicitly contains planning sections or structured categories such as:
  - “2025 Overall Planning”, “Income”, “Estate Planning”, “Investments”, “Insurance”, “Goals & Values”, “Family”, etc.
- A narrative agenda that contains blocks like:
  - Discussion, Q&A, Recommendations, Outcome, Service & Fees, Financial Planning Data, etc.
- Must include substantive planning or discussion content with **financial/strategic data**, not only logistics.

❌ **What is NOT an Agenda**:
- Administrative meeting notes (e.g., “Type of Meeting: Face to Face / Zoom”, “Who attended”, “Communication Updates”).
- Metadata such as dates, attendance, or communication details without financial or planning content.
- Any section explicitly labeled “No Recap Letter Needed” or equivalent.

⚠️ If the document does not explicitly contain agenda-style planning content, **do not classify it as an Agenda**.

Rules for extraction:
- Preserve all headers, subsections, and bullets exactly as written.
- Do not summarize, paraphrase, rewrite, condense, or expand.
- Do not infer missing sections or invent new content.
- Reproduce wording, formatting, bulleting, and punctuation exactly as in the source.
- If any section is empty in the source, keep it empty — do not remove.

2) **[Reserved Section Placeholder]**
- Always return this section, even if empty.
- Label must remain exactly as specified.
- Do not infer, summarize, or transform content.

⚠️ **ABSOLUTE PROHIBITIONS**:
- No summaries.
- No paraphrasing.
- No logical inferences.
- No new agenda “types” beyond **Narrative** or **Structured Planning**.
- No treating meeting metadata, attendance records, or communication updates as an agenda.
- No addition or removal of sections, headers, bullets, or notes.

✅ Output must be a **verbatim extraction** of the actual agenda content — *Narrative Agenda* or *Structured Planning Agenda* only.

2) **Topics Discussed**
- Expanded bullets (aim 5–7 if available), derived from the agenda content (narrative or structured).
- Keep granular and factual; do not speculate.
- Preserve names, roles, dollar amounts, tickers, and dates exactly as documented.
- If a detail is missing, write **"(no detail in notes)"**.

- Do **not** include other meetings; strictly the single most recent one per selection logic.

Output Structure (only these sections)
Most Recent Meeting — {client_name}

Date — Type: [DD-MMM-YYYY — Review A/B/C/D, with location tag if present; include original range text if applicable]

Agenda (Content):
EXACT AGENDA CONTENT HERE AS HOW IT IS (narrative or structured)

Topics Discussed:
- Bullet 1
- Bullet 2
- Bullet 3
- Bullet 4
- Bullet 5
- (continue as needed)

Formatting Rules
- Normalize dates in the header line to **DD-MMM-YYYY** (English months: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec).
- Use Markdown headings and bold labels exactly as shown.
- Keep Agenda (Content) and Topics strictly separated.

```
        - Do **not** include conclusions or recommendations.

        ## Trace Line (for auditability)
        At the very end, add a single plain line:
        Selected using: Meeting Date tolerance ±2 days around Target Meeting Date =
{(datetime.strptime(input['last_meeting_date'], "%Y-%m-%d") +
timedelta(days=1)).strftime("%Y-%m-%d")}; Reference Date = {today}.
        """

        response = model.invoke(prompt).content

        logger.info(f"Latest Agenda Info for Client: {response}")

        return {'latest_agenda_info': response}

    except Exception as e:
        logger.error(f"Error in retrieve_latest_agenda_info: {e}")
        raise

def generate_next_agenda_topics(
    input: RedtailGraph
):
    """
    Generates the next agenda topics for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        orm_instance_advisor = LookupClientAdvisorService()

        advisor_name = orm_instance_advisor.get_latest_record(
            client_name=client_name
        )['advisor_name']
```

```
    nickname_instance = LookupClientNicknameService()

    nickname_record = nickname_instance.get_latest_record(
        client_name=client_name
    )

    nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

    today = date.today().strftime("%B %d, %Y")

    redtail_names_lookup_service = RedtailNamesLookupService()

    result = redtail_names_lookup_service.get_latest_record(
        name=client_name
    )

    client_result = get_redtail_contact(
        input = RedtailContactSchema(
            **{
                "first_name": result['first_name'],
                "last_name": result['last_name']
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
            }
        )
    )

    notes = get_redtail_contact_notes(
        input = RedtailContactNotesSchema(
            **{
                "contact_id": str(client_result['id']),
                "start_date": input['last_meeting_date'],
                "end_date": datetime.now().strftime("%Y-%m-%d")
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
```

```
            }
        )
    )

    print("Notes: ", notes)

    prompt = f"""
    You are an analyst who generates the **Topics** section for the next client meeting agenda.

    You are given:
    - A chronological evolution of past meetings (agendas and notes).
    - The latest meeting details (Agenda and Topics).
    - The most recent advisor notes and updates (post-meeting).

    Your job is to create a set of **expanded bullet points (5–10 if possible)** that should be included in the **next meeting agenda** — but **NEVER** include items that have already been **completed/closed/resolved**. Completed items often appear in **Notes** (workflows, tasks, tickets) and **must be excluded** from Topics.

    ALL THE NEW TASKS MUST BE WITHIN THE TIME FRAME: {input['last_meeting_date']} to {datetime.now().strftime("%Y-%m-%d")}

    **Audience & Tone**
    - Audience: financial advisors, compliance officers, operations staff.
    - Output language: **English**.
    - Style: factual, neutral, precise.
    - No speculation, no personal advice, no extra commentary.

    **Inputs**
    - Client Name: {client_name}
    - Advisor Name: {advisor_name}
    - Historical Meeting Evolution: {input['all_past_meetings']}
    - Latest Meeting Information: {input['latest_agenda_info']}
    - Recent Advisor Notes (from {input['last_meeting_date']} to {datetime.now().strftime('%Y-%m-%d')}): {notes}
    - Today: {date.today().strftime("%B %d, %Y")}
    - Optional client nickname (may appear in sources): {nickname}

    ---

    ## **Strict Generation Rules**
```

1) **Only open/pending items.** Include **only** items that are currently **open, pending, in progress, needs decision, blocked, awaiting docs, follow-up required, due by <date>**, or equivalent states.

2) **Exclude completed/closed.** If an item is marked or implied as **done/complete/closed/resolved/implemented/fulfilled/satisfied**, or a workflow/activity is **completed** (e.g., “workflow ... completed on <date>”), **do not include it** as a Topic.

3) **Carry-over with status check.** Carry over from the latest meeting **only if** the item remains unresolved **and** is not marked completed in Notes.

4) **De-duplicate.** Merge duplicates across meetings/notes; keep the clearest, most recent wording.

5) **Preserve facts exactly.** Keep **names, roles, accounts, tickers, dollar amounts, dates, deadlines** exactly as documented.

6) **No conclusions/recommendations.** Provide **agenda topics only**; no analysis, advice, or conclusions.

7) **Missing specificity.** If a specific detail is missing but the item is clearly open, include the topic and write **"(no detail in notes)"** for the missing piece.

8) **Time awareness.** If a due date is **past** and the item is **not** marked completed, keep it as an open Topic and **retain the original date** (do not change it).

9) **Granularity.** Prefer 5–10 concise bullets that are **actionable** and **review-ready** (e.g., “Review X”, “Confirm Y by <date>”, “Obtain Z document from <name>”).

10) **Nickname handling.** If sources use {nickname}, normalize to the formal client name in the output unless quoting an entity name that officially uses the nickname.

Completed/Closed Detection (treat any of these as completed):

- Explicit markers: **Completed, Complete, Done, Resolved, Closed, Implemented, Satisfied, Fulfilled, Finalized**.
- Workflow/activity language: “**workflow ... completed**”, “**activity completed**”, “**ticket closed**”, “**case closed**”, “**issue resolved**”.
- Symbols/checkboxes: **[x]**, **✓**, **✔**, “**Done.**”
- Past-tense confirmation with no remaining action: “**has been processed/sent/filed/executed**” (and no follow-up requested).
- Phrases like: “**No further action**”, “**Nothing pending**”, “**All set**”.

If any of the above appears for an item (in past agendas or Notes), **exclude it** from Topics.

Open/Pending Detection (eligible for Topics):

- Phrases like **Pending, In progress, To do, Awaiting, Follow-up, Needs decision, Needs approval, Provide/Obtain/Send/Upload, Schedule/Reschedule**, “**Left to satisfy**”, “**Outstanding**”, “**Due <date>**”, “**Renew by <date>**”, “**Confirm by <date>**”.

- Items mentioned in the latest meeting that have **no completion evidence** in Notes.

Output Quality Requirements
- Each bullet must be a **single concise line** focused on the action/review item (include entity names, accounts, tickers, $ amounts, dates as documented).
- Use consistent verbs: **Review / Confirm / Obtain / Provide / Schedule / Verify / Reconcile / Update / Decide**.
- Do **not** restate history; make the bullet forward-looking for the next meeting.
- If nothing remains open, write a single bullet: **"(no open agenda topics identified in notes)"**.

Required Output Structure

Next Meeting Agenda — {client_name}

Topics:
- Bullet 1
- Bullet 2
- Bullet 3
- Bullet 4
- Bullet 5
- (continue as needed)

Formatting Rules
- Use Markdown headings and bold labels exactly as shown.
- Keep all content as factual bullets.
- Do not include an Agenda (Content) section — only **Topics** for the next meeting.

Implementation Hints (for you, not to output)
- Treat any item with an explicit completion date/time or completion phrase in Notes as **completed** → exclude.
- When two items conflict (one says open, a newer note says completed), trust the **newer** source.
- Normalize dates as they appear in the sources; **do not** invent new dates or adjust wording.
"""

response = model.invoke(prompt).content

logger.info(f"Next Agenda Topics for Client: {response}")

prompt = f"""
You are an agenda analyst tasked with building the **validated Topics list** for the next client meeting.

You are given three inputs:
1. The current list of **Agenda Topics** (from prior agendas):
{response}
2. The **Latest Agenda** (most recent version before this validation).
{input['latest_agenda_info']}
3. The most recent **Advisor Notes** (activities, workflows, tasks, emails).
{notes}

ALL THE NEW TASKS MUST BE WITHIN THE TIME FRAME: {input['last_meeting_date']} to {datetime.now().strftime("%Y-%m-%d")}

Your job:
- **Re-validate** all agenda items.
- **Suppress/remove** any item that is already **completed/closed/resolved** according to the Notes.
- **Keep** items that are still **open, pending, or unresolved**.
- **Add** new items from the Notes if they are clearly open/pending but not already in the Topics list.

Strict Rules

1. Completed = Exclude (field-aware, HTML/Markdown aware)
Mark as **COMPLETED → EXCLUDE** if any of the following **hard signals** are present:

- **Field `COMPLETED DATE` exists** and is not empty in the activity/notes.
- Explicit completion markers in **plain text, HTML, or Markdown**, for example:
- `<h5>… was completed</h5>`
- `**Activity completed**`, `_Workflow completed_`
- Table rows like `<td>COMPLETED DATE</td><td>June 25 2025 …</td>`
- Markdown tables with a “COMPLETED DATE” cell
- Keywords (case-insensitive): `was completed`, `completed`, `done`, `resolved`, `closed`, `finalized`, `implemented`, `fulfilled`, `satisfied`, `all set`, `no further action`.
- Confirmations such as: `You have successfully completed…`.

Reopen logic:

- If evidence exists *after* the `COMPLETED DATE` that the task was **reopened** (phrases: `reopen`, `pending again`, `follow-up required`, `left to satisfy`, `due <date>`), then treat as **open** and include it with the most recent wording.
- Otherwise, exclude it.

2. Latest Agenda Deduplication
- Do **not** repeat Latest Agenda items if Notes confirm they are completed.
- Carry them forward only if there is **no evidence of completion**.

3. Open/Pending = Keep or Add
- Include Topics if Notes show: `Pending`, `In progress`, `To do`, `Awaiting`, `Follow-up`, `Needs decision`, `Outstanding`, `Left to satisfy`, `Due <date>`.
- If such an item is not in the Agenda Topics list yet, **add it**.

4. De-duplicate
- Normalize text across plain text, HTML, and Markdown (strip tags, decode entities, flatten formatting).
- Merge duplicates across Agenda Topics, Latest Agenda, and Notes.
- Use SUBJECT or normalized equivalence (`401(k)` ≈ `401k`, possessives, minor variants).
- Keep the clearest, most recent wording.

5. Preserve factual detail
- Copy names, accounts, tickers, amounts, and dates **exactly as written**.
- Do not invent, estimate, or summarize.

6. Output format
- Audience: financial advisors, compliance officers, operations staff.
- Language: English.
- Style: factual, neutral, precise.
- No speculation, advice, or commentary.

Output Structure

Validated Agenda Topics — {client_name}

Topics:
- Bullet 1 (open or newly added)
- Bullet 2
- Bullet 3
- (continue as needed)

```
        If there are no open/pending items left:
        - **"(no open agenda topics identified in notes)"**

        ---

        ## **Implementation Hints (for you, not to output)**
        - Normalize HTML and Markdown first → strip tags, decode `&#39;`, flatten formatting symbols (`**bold**`, `_italic_`).
        - Notes are the **source of truth** for completion.
        - Always cross-check Topics, Latest Agenda, and Notes together.
        - New bullets should use action verbs: **Review / Confirm / Obtain / Provide / Schedule / Verify / Update / Decide**.
        - Keep bullets short, factual, forward-looking.
        - When in doubt, **exclude** (better to under-include than to repeat completed work).
        """

        response = model.invoke(prompt).content

        logger.info(f"Next Agenda Topics for Client (Refined): {response}")

        return {'next_agenda_topics': response}

    except Exception as e:
        logger.error(f"Error in generate_next_agenda_topics: {e}")
        raise

def build_latest_client_info(
    input: RedtailGraph
):
    """
    Builds the latest client info for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])
```

```
    orm_instance_advisor = LookupClientAdvisorService()

    advisor_name = orm_instance_advisor.get_latest_record(
        client_name=client_name
    )['advisor_name']

    nickname_instance = LookupClientNicknameService()

    nickname_record = nickname_instance.get_latest_record(
        client_name=client_name
    )

    nickname = nickname_record['client_nickname'] if nickname_record and 'client_nickname'
in nickname_record else None

    today = date.today().strftime("%B %d, %Y")

    redtail_names_lookup_service = RedtailNamesLookupService()

    result = redtail_names_lookup_service.get_latest_record(
        name=client_name
    )

    client_result = get_redtail_contact(
        input = RedtailContactSchema(
            **{
                "first_name": result['first_name'],
                "last_name": result['last_name']
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
            }
        )
    )

    notes = get_redtail_contact_notes(
        input = RedtailContactNotesSchema(
            **{
                "contact_id": str(client_result['id']),
                "start_date": input['last_meeting_date'],
```

```
                "end_date": datetime.now().strftime("%Y-%m-%d")
            }
        ),
        account = RedtailAccountCredentialsSchema(
            **{
                "username": os.environ.get("RT_USERNAME"),
                "password": os.environ.get("RT_PASSWORD")
            }
        )
    )

    prompt = f"""
    You are an analyst responsible for producing an **up-to-date and fully refreshed client profile**.
    You are given:
    - The **Initial Client Profile** (for guidance only, not for copying).
    - The **Full Meeting Notes** (historical and most recent).
    - The **Next Topics** for the upcoming agenda.

    YOU MUST CONSIDER THIS TIME FRAME: {input['last_meeting_date']} to {datetime.now().strftime("%Y-%m-%d")}

    Your tasks are:

    ---

    ### 1. Build the Updated Client Profile
    - Ignore the initial profile as a baseline — use it only for orientation.
    - Reconstruct a **completely updated profile** using the latest available notes and next topics.
    - Every attribute must be updated to the newest data available.
    - If information is missing, write **"(no detail in notes)"**.
    - For each attribute, provide:
    - The **current value** (factually documented).
    - A short **"> Reasoning:"** line explaining the source (meeting date, note, or explicit statement).

    The updated profile must cover these sections:

    #### 🗒️ Demographics
    - **Full Legal Name**:
    > Reasoning:
    - **Gender**:
    > Reasoning:
```

👨‍👩‍👧‍👦 Family & Relationship Context
- **Marital Status**:
> Reasoning:
- **Spouse/Partner Name**:
> Reasoning:
- **Spouse/Partner DOB**:
> Reasoning:
- **Children / Grandchildren**:
> Reasoning:
- **POAs (Power of Attorney)**:
> Reasoning:
- **Death Dates (Spouse / Parent / IRA Originator)**:
> Reasoning:

⚠️ Special Conditions or Flags
- **Disability / Terminal Illness / LTC Status**:
> Reasoning:
- **Veteran or Military Status**:
> Reasoning:
- **Non-citizen or Immigration Flag**:
> Reasoning:
- **Legal Custodianship / Special Needs**:
> Reasoning:

🧠 Client Philosophy & Behavioral Markers
- **Legacy or Inheritance Goals**:
> Reasoning:
- **Risk Tolerance / Investment Beliefs**:
> Reasoning:
- **Charitable Giving Intentions**:
> Reasoning:
- **Ethical or ESG Constraints**:
> Reasoning:

🔍 Employment & Retirement Data

- **Employment Status (Current)**:
> Reasoning:
- **Retirement Intentions / Timeline**:
> Reasoning:
- **Employer Plan Participation (401k, pension, benefits)**:
> Reasoning:
- **Medicare Deferral or Trigger Context**:
> Reasoning:
- **Any Mention of Exit Planning or Buyouts**:
> Reasoning:
- **Work Status Impacting Other Areas (e.g., tax, savings, insurance)**:
> Reasoning:

📅 Meeting & Engagement Profile
- **Last Documented Meeting or Review Date**:
> Reasoning:
- **Meeting Frequency / Cadence (e.g., quarterly, annual)**:
> Reasoning:
- **Client Meeting Style or Review Type** (A, B, C, D Reviews):
> Reasoning:
- **Topics Typically Covered in Their Reviews**:
> Reasoning:
- **Engagement Style** (e.g., responsive, passive, skips follow-ups):
> Reasoning:
- **Any Missed, Postponed, or Skipped Reviews**:
> Reasoning:
- **Notes on Preferences or Patterns**:
> Reasoning:

---

2. Expand Information for Each Next Topic
- For every **Next Topic**, provide a detailed section with all available supporting data from the latest notes/meetings.
- Preserve exact values (amounts, dates, account numbers, tickers, percentages).
- If missing, write **"(no detail in notes)"**.
- ⚠️ Each topic expansion must contain **5–8 complete sentences** of domain-specific information.
- Each sentence must be a full statement in bullet-point format, anchored in concrete facts (numbers, percentages, ages, plan types).
- Optional: Add a **Traceability** line at the end of each topic showing the meeting/note source.

Audience & Tone
- Audience: financial advisors, compliance officers, operations staff.
- Output language: **English**.
- Style: factual, structured, precise.
- No speculation, no commentary, no conclusions.

Inputs
- Client Name: {client_name}
- Initial Client Profile: {input['client_profile']}
- Meeting Notes: {notes}
- Next Topics: {input['next_agenda_topics']}

Required Output Structure

Updated Client Profile — {client_name}

[All attributes completed with values + reasoning as specified above]

Next Meeting Topics — Expanded Details

Topic 1: [Exact Topic Text]
- Supporting detail 1
- Supporting detail 2
...
- Supporting detail N
Traceability: [Meeting/Note source]

Topic 2: [Exact Topic Text]
- Supporting detail 1
- Supporting detail 2
...
- Supporting detail N
Traceability: [Meeting/Note source]

(continue for all topics)

```
        """

        content = model.invoke(prompt).content

        logger.info(f"Latest Client Info for Client: {content}")

        return {'latest_client_info': content}

    except Exception as e:
        logger.error(f"Error in build_latest_client_info: {e}")
        raise

def extract_external_account_info(
    input: RedtailGraph
):
    """
    Extracts the external account info for a given client name.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        household_service = HouseholdLookupService()
        client_name = household_service.find_household_by_individual(input['client_name'])

        prompt = f"""
        # Role
        You are a **Financial Data Extraction Agent** specialized in identifying and normalizing account-level information within financial planning documents (Redtail notes, meeting summaries, CRM exports, or agenda data).

        # Objective
        Extract **ONLY CONFIRMED external accounts with explicit details** — meaning any accounts **held outside the advisor-managed or custodian-integrated ecosystem** (e.g., held-away accounts, self-managed brokerage, retirement, or bank accounts not under direct management).
```

CRITICAL RULE: If the text says "Not indicated", "Not mentioned", "No explicit mention", or any similar phrase indicating ABSENCE of information, DO NOT extract that account. Only extract accounts with POSITIVE confirmation and concrete details.

EMPLOYER-SPONSORED PLANS = EXTERNAL ACCOUNTS:
Employer-sponsored retirement plans are considered EXTERNAL ACCOUNTS, including:
- 401(k), 403(b), 457(b), 401(a), SIMPLE 401(k), and any other employer-sponsored plans, etc.

IRAs are NOT external accounts:
- Traditional IRA, Roth IRA, SEP IRA, SIMPLE IRA are NOT external accounts
- These are internal accounts managed within the advisory relationship
- Exception: If an IRA is explicitly stated as "held outside of advisory" or "external", then include it

Instructions
1. **Search for key phrases** including but not limited to:
- "external account", "held-away", "outside account", "outside asset", "non-managed account"
- **Employer-sponsored retirement plans**: 401(k), 403(b), 457(b), 401(a), SIMPLE 401(k), etc.
- **DO NOT include IRAs by default**: Traditional IRA, Roth IRA, SEP IRA, SIMPLE IRA are NOT external accounts (unless explicitly stated as "external" or "held away")
- references to custodians or financial institutions **not** matching the advisor's integrated systems
- self-directed or third-party investment accounts, savings, or bank accounts

2. **For each mention found**, extract a structured entry with the following attributes **with Chain of Thought reasoning for each field**:

- **Account Type**: (e.g., 401(k), 403(b), 457(b), 401(a), SIMPLE 401(k), etc.) - Employer-sponsored plans are external accounts. IRAs (Traditional, Roth, SEP, SIMPLE IRA) are NOT external unless explicitly stated.
  > Reasoning: [Explain where/how this was identified, or why it's N/A]

- **Custodian / Institution**: (e.g., Voya, Vanguard, Thrivent, Local Credit Union, etc.)
  > Reasoning: [Explain where/how this was identified, or why it's N/A]

- **Owner(s)**: (e.g., Mark Patterson, Marcie Patterson, Joint)
  > Reasoning: [Explain where/how this was identified, or why it's N/A]

- **Balance or Value (if available)**: (include $ amounts if mentioned) => BALANCES ONLY FOR EXTERNAL ACCOUNTS: 401(k), 403(b), 457(b), 401(a), SIMPLE 401(k), and other employer-sponsored plans, etc. NOT for IRAs unless explicitly external.

> Reasoning: [Explain where/how this was identified, or why it's N/A]

- **Status / Context**: (e.g., rollover initiated, contribution ongoing, held-away, needs linking, inactive, etc.)
> Reasoning: [Explain where/how this was identified, or why it's N/A]

- **Source Section**: (attribute or note where found — e.g., `all_past_meetings`, `next_steps`, `client_profile`)

- **Evidence Snippet**: short direct quote (≤25 words) showing the context of the mention.

⚠️ **Important**:
- You are NOT required to fill all attributes for each account.
- **Each field must include a reasoning line** that explains either where the information was found OR why it's marked as N/A.
- This Chain of Thought (CoT) approach prevents hallucinations by grounding each value in explicit evidence.

- **ABSOLUTE PROHIBITION - DO NOT EXTRACT ACCOUNTS WHEN**:
  - Text says "Not indicated", "Not mentioned", "No explicit mention", "No specific details"
  - Reasoning line says "no explicit mention", "not provided", "not indicated"
  - Speculative language: "may have", "possibly", "might be", "could have"
  - Vague references without concrete institution names or account numbers
  - ONLY section headers exist without actual account data (e.g., "401k, pension, benefits: Not indicated")

- **ONLY EXTRACT when ALL of these exist**:
  - Specific account type explicitly mentioned (e.g., "401(k)", "403(b)", "457(b)", "401(a)", "SIMPLE 401(k)", etc.) - NOT IRAs by default
  - At least one additional concrete detail: institution name, balance, contribution amount, or owner name
  - NO negative language in the source text (no "not indicated", "not mentioned", etc.)
  - **NOTE**: For employer-sponsored plans (401(k), 403(b), 457(b), 401(a), SIMPLE 401(k), etc.), these are ALWAYS external accounts - just need the plan type + one concrete detail
  - **IRAs are NOT external by default**: Do NOT extract Traditional IRA, Roth IRA, SEP IRA, SIMPLE IRA unless explicitly stated as "external" or "held away"

3. **Strict Evidence Requirements**:
- ONLY include external accounts when there is an explicit quote or clear statement with concrete details
- **IF YOU SEE ANY OF THESE PHRASES, DO NOT EXTRACT THE ACCOUNT**:
  - "Not indicated", "Not mentioned", "Not specified", "Not provided"
  - "No explicit mention", "No specific details", "No information"

```
        - "There is no explicit mention of..."
        - Any reasoning that says information is absent or not provided
      - **For EMPLOYER-SPONSORED PLANS** (401(k), 403(b), 457(b), 401(a), SIMPLE 401(k), etc.):
        - These are ALWAYS external accounts
        - Extract when: plan type + at least one concrete detail (owner, balance, institution, or contribution amount)
        - DO NOT extract if text says "Not indicated" or similar absence language
        - **DO NOT extract IRAs by default**: Traditional IRA, Roth IRA, SEP IRA, SIMPLE IRA are NOT external accounts unless explicitly stated as "external" or "held away"
      - **For OTHER external accounts**, infer held-away status ONLY when:
        - A specific custodian name (not internal) is explicitly mentioned WITH positive confirmation
        - "Rollover from [Specific Institution Name]" is stated with the actual institution name
        - Account details (type, balance, institution) are explicitly provided with positive language
      - DO NOT infer based solely on job titles, employment status, or general statements without specific account details
      - **FINAL CHECK**: Before extracting any account, verify the source text contains POSITIVE confirmation, not absence of information

      # Null-Detection Policy
      If no explicit or inferable external accounts are found, do not fabricate or infer any.
      Simply state that no external accounts were identified in the provided information.
      Avoid listing or describing accounts unless they are explicitly supported by evidence.

      Context:
      Client Profile: {input['client_profile']}
      Employment and Meeting Insights: {input['employment_and_meeting_insights']}
      """

      response = model.invoke(prompt).content

      logger.info(f"External Account Info for Client: {response}")

      return {'external_account_info': response}

  except Exception as e:
      logger.error(f"Error in extract_external_account_info: {e}")
      raise

def merge_redtail_node(state: RedtailGraph):
  return state

def build_redtail_graph():
```

```
    """
    Builds a graph for a given client name.
    """

    # Create the Graph
    graph_builder = StateGraph(RedtailGraph)

    graph_builder.add_node("get_client_age_node", get_client_age)
    graph_builder.add_node("build_client_profile_node", build_client_profile)
    graph_builder.add_node("build_employment_and_meeting_insights_node",
build_employment_and_meeting_insights)
    graph_builder.add_node("build_thresold_evaluation_for_client_node",
build_thresold_evaluation_for_client)
    graph_builder.add_node("extract_latest_meeting_insights_node",
extract_latest_meeting_insights)
    graph_builder.add_node("extract_next_steps_node", extract_next_steps)
    graph_builder.add_node("timeline_and_milestone_trigger_node",
timeline_and_milestone_trigger)
    graph_builder.add_node("extract_all_past_meetings_node", extract_all_past_meetings)
    graph_builder.add_node("retrieve_latest_agenda_info_node", retrieve_latest_agenda_info)
    graph_builder.add_node("generate_next_agenda_topics_node",
generate_next_agenda_topics)
    graph_builder.add_node("build_latest_client_info_node", build_latest_client_info)
    graph_builder.add_node("extract_external_account_info_node",
extract_external_account_info)
    graph_builder.add_node("merge_redtail_node", merge_redtail_node, defer=True)

    graph_builder.add_edge(START, "get_client_age_node")
    graph_builder.add_edge("get_client_age_node", "build_client_profile_node")
    graph_builder.add_edge("get_client_age_node",
"build_employment_and_meeting_insights_node")
    graph_builder.add_edge("get_client_age_node", "build_thresold_evaluation_for_client_node")
    graph_builder.add_edge("build_client_profile_node", "extract_latest_meeting_insights_node")
    graph_builder.add_edge("build_employment_and_meeting_insights_node",
"extract_latest_meeting_insights_node")
    graph_builder.add_edge("build_thresold_evaluation_for_client_node",
"extract_latest_meeting_insights_node")
    graph_builder.add_edge("extract_latest_meeting_insights_node", "extract_next_steps_node")
    graph_builder.add_edge("extract_latest_meeting_insights_node",
"timeline_and_milestone_trigger_node")

    graph_builder.add_edge("extract_next_steps_node", "extract_all_past_meetings_node")
    graph_builder.add_edge("extract_next_steps_node", "retrieve_latest_agenda_info_node")
    graph_builder.add_edge("timeline_and_milestone_trigger_node", "merge_redtail_node")
```

```
    graph_builder.add_edge("extract_all_past_meetings_node",
"generate_next_agenda_topics_node")
    graph_builder.add_edge("retrieve_latest_agenda_info_node",
"generate_next_agenda_topics_node")
    graph_builder.add_edge("generate_next_agenda_topics_node",
"build_latest_client_info_node")
    graph_builder.add_edge("generate_next_agenda_topics_node",
"extract_external_account_info_node")
    graph_builder.add_edge("build_latest_client_info_node", "merge_redtail_node")
    graph_builder.add_edge("extract_external_account_info_node", "merge_redtail_node")

    graph_builder.add_edge("merge_redtail_node", END)

    # Compile the Graph
    graph = graph_builder.compile()

    return graph
```

```
import decimal
import os
import time
from typing import Annotated, Any, Dict, TypedDict
from langchain_openai import AzureChatOpenAI
import psycopg2
from pydantic import BaseModel, Field

from api.app.agenda_generator.v2.redtail_graph import ApplicabilityToClient
from api.app.agenda_generator.v2.visual_agenda.schemas.utils import RMDsUtil
from api.app.agents.data_warehouse.agents.data_warehouse_fidelity_wealthscape import
data_warehouse_fidelity_wealthscape_agent
from api.app.agents.data_warehouse.tools.data_warehouse_queries.black_diamond import
get_transactions
from api.app.agents.data_warehouse.tools.data_warehouse_queries.fidelity_wealthscape
import ask_sql_query, get_icp_traces, get_transactions_reusable
from api.app.agents.data_warehouse.tools.data_warehouse_tools import
dw_transaction_tool_agenda_generator
from api.app.config.event_stream_helper import emit_message
from api.app.config.logger import logger
from api.app.helpers.azure_postgresql_manager import AzurePostgreSQLManager,
HouseholdLookupService, LookupClientAdvisorService
from langchain_core.output_parsers import JsonOutputParser
```

```
from
api.app.agents.data_warehouse.tools.data_warehouse_queries.fidelity_functions.targets
import get_target_vs_holdings_comparison_core
from
api.app.agents.data_warehouse.tools.data_warehouse_queries.fidelity_functions.strategi
es import get_strategies

from langgraph.graph import StateGraph, END, START
import concurrent.futures
from typing import List



from decimal import Decimal
from datetime import datetime, date
from typing import Optional
import os

from dotenv import load_dotenv, find_dotenv

from api.app.helpers.helper_orm import MegaTableManager

load_dotenv(find_dotenv(), override=True)

def parse_builtins(obj):
   if isinstance(obj, Decimal):
       return float(obj)
   elif isinstance(obj, (datetime, date)):
       return obj.isoformat()
   elif isinstance(obj, dict):
       return {k: parse_builtins(v) for k, v in obj.items()}
   elif isinstance(obj, list):
       return [parse_builtins(i) for i in obj]
   return obj

def filter_transactions_basic(data):
   return [
       {
           "account_number": item.get("account_number"),
           "account_owner_name": item.get("account_owner_name"),
           "transaction_type": item.get("transaction_type"),
           "transaction_date": item.get("trade_date_key"),
           "value": item.get("net_amount_cash"),
```

```
        }
        for item in data
    ]

def filter_transactions_basic_bd_v2(data):
    return [
        {
            "account_number": item.get("account_number"),
            "account_owner_name": item.get("name"),
            "transaction_type": item.get("transaction_type"),
            "transaction_date": str(item.get("trade_date")),
            "value": float(item.get("market_value", 0.0)),
        }
        for item in data
    ]

def get_transactions_core_v2(
    user_name: Optional[str] = None,
    account_number: Optional[str] = None,
    account_type: Optional[str] = None,
    date_from: Optional[str] = None,
    date_to: Optional[str] = None,
    transaction_type: Optional[str] = None,
    limit: Optional[int] = 30,
    db_name: str = None
):
    filters = []
    params = []
    if user_name:
        user_name_clean = user_name.strip().lower()
        like_pattern = f"%{'%'.join(user_name_clean.split())}%"
        filters.append(
            "(LOWER(da.account_owner_name) LIKE %s OR "
            "LOWER(da.short_name) LIKE %s OR "
            "LOWER(da.account_note_line_1) LIKE %s OR "
            "LOWER(da.account_note_line_2) LIKE %s OR "
            "LOWER(da.account_note_line_3) LIKE %s)"
        )
        params.extend([like_pattern] * 5)
    if account_number:
        account_number_clean = account_number.strip()
        filters.append("da.account_number = %s")
```

```
        params.append(account_number_clean)
    if account_type:
        account_type_clean = account_type.strip()
        filters.append("da.account_type = %s")
        params.append(account_type_clean)
    if transaction_type:
        filters.append("ft.transaction_type = %s")
        params.append(transaction_type)
    if date_from:
        try:
            from_key = int(date_from.replace("-", ""))
            filters.append("ft.entry_date_key >= %s")
            params.append(from_key)
        except Exception:
            pass
    if date_to:
        try:
            to_key = int(date_to.replace("-", ""))
            filters.append("ft.entry_date_key <= %s")
            params.append(to_key)
        except Exception:
            pass
    where_clause = ""
    if filters:
        where_clause = "WHERE " + " AND ".join(filters)
    sql_script = f"""
    SELECT
        da.account_number,
        da.account_type,
        da.account_owner_name,
        da.short_name,
        da.account_note_line_1,
        da.account_note_line_2,
        da.account_note_line_3,
        ds.symbol,
        ds.description,
        ft.*
    FROM fact_transaction ft
    JOIN dim_account da ON ft.account_key = da.account_key
    LEFT JOIN dim_security ds ON ft.security_key = ds.security_key
    {where_clause}
    ORDER BY ft.entry_date_key DESC
```

```
    """
    if limit is not None and isinstance(limit, int) and limit > 0:
        sql_script += " LIMIT %s"
        params.append(limit)
    sql_script += ";"

    raw_results = ask_sql_query(sql_script, params=params, db_name=db_name or
os.getenv("DW_PGDATABASE_FIDELITY_WEALTHSCAPE"))
    parsed_results = [parse_builtins(r) for r in raw_results]
    return filter_transactions_basic(parsed_results)

def format_cash_flow_data(icp_data: dict):

    try:
        prompt = f"""
        You are a financial assistant specialized in presenting ICP (Integrated
Cashiering Platform) transaction data as clean, structured cash flow tables.

        Your task is to analyze the list of transactions provided and return:

        1. **A table of receipts (inflows)**:
        - Columns: Date, Amount (USD), From, To, Type, Match Status
        - Group by: inflows (where 'recv_icp_id' is not null)

        2. **A table of disbursements (outflows)**:
        - Columns: Date, Amount (USD), From, To, Type, Status
        - Group by: outflows (where 'disb_icp_id' is not null)

        3. **A summary block** at the end with:
        - Total Inflows
        - Total Outflows
        - Net Flow (Inflows - Outflows)

        **Formatting rules:**
        - Dates should be in `MM/DD/YYYY` format.
        - Amounts should be formatted as `$#,###.00 USD`.
        - Align all rows properly in table format.
        - Only include fields relevant to inflow or outflow; avoid empty/null fields.
        - Group and sort transactions by date ascending.

        Input data will be a list of dictionaries, each representing a transaction with
keys like:
```

```
      - `txn_date`, `txn_amount`, `recv_icp_id`, `recv_account`, `recv_from_account`,
`recv_type`, `match_status`, `disb_icp_id`, `disb_account`, `disb_to_account`,
`disb_type`, `status`

      Your output should include two tables (inflows and outflows) and a summary
section.
      Only use the fields that are relevant to each transaction type.
      Do not include transactions with neither inflow nor outflow direction.

      Return the output as Markdown or plain text table, not raw JSON.

      Here you have the Cash Flow Data:
      {icp_data}
      """

      model = AzureChatOpenAI(
          azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
          api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
          temperature=0,
          max_tokens=None,
          timeout=None,
          max_retries=2
      )

      result = model.invoke(prompt).content

      logger.info(f"Cash Flow Data: {result}")

      return result
   except Exception as e:
      logger.error(f"Error in format_cash_flow_data: {e}")
      raise

def format_transactions(transactions: dict, totals_transactions: dict):
   try:
      prompt = f"""
      You are a financial transaction analyst. You are given a list of raw
transaction records. Each includes:

      - `account_number`: Account ID
      - `account_owner_name`: Name of the account holder
```

```
        - `transaction_type`: Free-text description of the transaction (e.g., "IRA
CONTRIBUTION", "TRANSFER", "PARTIAL WITHDRAWAL")
        - `entry_date`: Transaction date (can be in `YYYYMMDD` or `YYYY-MM-DD`)
        - `net_amount_cash`: Amount of the transaction

        ---

        THIS MUST BE YTD: {datetime.now().replace(month=1, day=1).strftime('%Y-%m-%d')}
to {datetime.now().strftime('%Y-%m-%d')}

        ### 🎯 OBJECTIVE

        You must process and output the data in **one section only**:


        ## 📊 YTD TOTALS PER ACCOUNT (As of {datetime.now().strftime('%Y-%m-%d')})

        For **each account**, calculate totals only using transactions **between
2025-01-01 and 2025-07-08**, inclusive.

        - Include only **Withdrawal** and **Contributions (Deposit)** categories
        - Always present results for each account (even if $0.00)

        Output for each account:

        ### YTD Totals — Account: <account_number> — <account_owner_name>

        * Total Contributions (Deposit): $X,XXX.XX
        * Total Withdrawals: $X,XXX.XX


        ---

        ## 📌 Final YTD Totals (Across All Accounts)

        At the end, present a **single combined total**:


        ### Final YTD Totals

        * Total Contributions (Deposit): $X,XXX.XX
        * Total Withdrawals: $X,XXX.XX
```

```
      HERE IS THE TOTALS FOR THE TRANSACTIONS: {totals_transactions}

      ---

      ### 🔍 CHAIN-OF-THOUGHT GUIDELINES

      - Normalize all dates → `YYYY-MM-DD`
      - Classify strictly per keyword logic:
          - “WITHDRAWAL” / “DISTRIBUTION” → Withdrawal
          - “CONTRIBUTION” / “DEPOSIT” → Contributions (Deposit)
      - Ensure **no YTD transaction is omitted** in the totals, even if not shown in
the table
      - ❗Do not add summaries, explanations, or extra notes — only formatted outputs
as described

      ---

      Here is the transaction list:
      {transactions}
      """

      model = AzureChatOpenAI(
          azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
          api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
          temperature=0,
          max_tokens=None,
          timeout=None,
          max_retries=2
      )

      result = model.invoke(prompt).content

      logger.info(f"Transactions: {result}")

      return result
  except Exception as e:
      logger.error(f"Error in format_transactions: {e}")
      raise

def convert_decimal_to_float(obj):
```

```
    if isinstance(obj, list):
        return [convert_decimal_to_float(i) for i in obj]
    elif isinstance(obj, dict):
        return {k: convert_decimal_to_float(v) for k, v in obj.items()}
    elif isinstance(obj, decimal.Decimal):
        return float(obj)
    else:
        return obj

def get_icp_traces_v2(
    user_name: Optional[str] = None,
    account_number: Optional[str] = None,
    date_from: Optional[str] = None,
    date_to: Optional[str] = None,
    db_name: str = None,
    limit: int = 100
):
    """
    Retrieve ICP traces using a full outer join between disbursement and receipt transactions for accounts matching the user_name or account_number, and optional date range.
    Filters by user_name (case-insensitive, partial match in account fields), account_number, and date range (date_from, date_to as YYYY-MM-DD, applies to transaction_date_key as YYYYMMDD int).
    Returns joined disbursement/receipt info, match status, and flow direction.
    limit: (Optional) The maximum number of results to return. Default is 100. You can override this by specifying a different limit.
    """
    user_name_filter = ""
    user_name_params = []
    if user_name:
        user_name_clean = user_name.strip().lower()
        like_pattern = f"%{'%'.join(user_name_clean.split())}%"
        user_name_filter = (
            "LOWER(account_owner_name) LIKE %s OR "
            "LOWER(short_name) LIKE %s OR "
            "LOWER(account_note_line_1) LIKE %s OR "
            "LOWER(account_note_line_2) LIKE %s OR "
            "LOWER(account_note_line_3) LIKE %s"
        )
        user_name_params = [like_pattern] * 5
```

```
    account_number_filter = ""
    account_number_params = []
    if account_number:
        account_number_clean = account_number.strip()
        account_number_filter = "account_number = %s"
        account_number_params = [account_number_clean]

    subquery_filters = []
    subquery_params = []
    if user_name_filter:
        subquery_filters.append(f"({user_name_filter})")
        subquery_params.extend(user_name_params)
    if account_number_filter:
        subquery_filters.append(f"({account_number_filter})")
        subquery_params.extend(account_number_params)
    subquery_where_sql = " AND ".join(subquery_filters)
    if subquery_where_sql:
        subquery_where_sql = f"WHERE {subquery_where_sql}"

    date_range_sql = ""
    date_range_params = []
    if date_from:
        try:
            from_key = int(date_from.replace("-", ""))
            date_range_sql += " AND transaction_date_key >= %s"
            date_range_params.append(from_key)
        except Exception:
            pass
    if date_to:
        try:
            to_key = int(date_to.replace("-", ""))
            date_range_sql += " AND transaction_date_key <= %s"
            date_range_params.append(to_key)
        except Exception:
            pass

    disb_params = subquery_params + date_range_params
    recv_params = subquery_params + date_range_params

    disb_like = '%disbursement%'
    recv_like = '%receipt%'
```

```
   sql_script = f"""
   SELECT
       COALESCE(d.disb_icp_id, r.recv_icp_id) AS icp_id,
       COALESCE(d.txn_date, r.txn_date) AS txn_date,
       COALESCE(d.txn_amount, r.txn_amount) AS txn_amount,
       d.disb_icp_id,
       d.disb_account,
       d.disb_to_account,
       d.disb_type,
       d.status,
       r.recv_icp_id,
       r.recv_account,
       r.recv_from_account,
       r.recv_type,
       CASE
           WHEN d.disb_icp_id IS NOT NULL AND r.recv_icp_id IS NOT NULL THEN 'Matched'
           WHEN d.disb_icp_id IS NOT NULL THEN 'Disbursement only'
           WHEN r.recv_icp_id IS NOT NULL THEN 'Receipt only'
           ELSE 'Unknown'
       END AS match_status,
       COALESCE(d.disb_account, '') || ' → ' || COALESCE(r.recv_account,
d.disb_to_account) AS flow_direction
   FROM (
       SELECT
           icp_id AS disb_icp_id,
           transaction_date_key AS txn_date,
           transaction_amount AS txn_amount,
           transaction_status as status,
           account_key::TEXT AS disb_account,
           REGEXP_REPLACE(sender_payee_account_name, '-', '', 'g') AS disb_to_account,
           transaction_description AS disb_type
       FROM fact_icp
       WHERE LOWER(transaction_description) LIKE %s
         AND account_key IN (
               SELECT account_key
               FROM dim_account
               {subquery_where_sql}
         ){date_range_sql}
   ) d
   FULL OUTER JOIN (
       SELECT
           icp_id AS recv_icp_id,
```

```
          transaction_date_key AS txn_date,
          transaction_amount AS txn_amount,
          account_key::TEXT AS recv_account,
          REGEXP_REPLACE(sender_payee_account_name, '-', '', 'g') AS
recv_from_account,
          transaction_description AS recv_type
      FROM fact_icp
      WHERE LOWER(transaction_description) LIKE %s
        AND account_key IN (
              SELECT account_key
              FROM dim_account
              {subquery_where_sql}
        ){date_range_sql}
  ) r
    ON d.txn_date = r.txn_date
   AND d.txn_amount = r.txn_amount
   AND d.disb_account = r.recv_from_account
   AND r.recv_account = d.disb_to_account
  ORDER BY txn_date
  """

  limit_param = []
  if limit is not None and isinstance(limit, int) and limit > 0:
      sql_script += "\nLIMIT %s"
      limit_param.append(limit)

  sql_script += ";"

  final_params = [disb_like] + disb_params + [recv_like] + recv_params + limit_param

  result = ask_sql_query(
      sql_script,
      params=final_params,
      db_name=db_name or os.getenv("DW_PGDATABASE_FIDELITY_WEALTHSCAPE")
  )

  return convert_decimal_to_float(result)

class AccountNumbers(BaseModel):
  account_numbers: List[str] = Field(..., description="The account numbers for the
client. Only the account numbers, no other information.")
```

```
class CustodiansGraph(TypedDict):
   client_name: str

   account_numbers: str
   account_number_list: Any

   account_balances: str
   account_balances_json: Any

   positions: str

   transactions: Any
   formatted_transactions: str

   formatted_transactions_prior_year: str

   totals_transactions: Any
   prior_year_total_transactions: Any

   totals_transactions_with_tax: Any
   prior_year_total_transactions_with_tax: Any

   rmds: str
   formatted_rmds: Any
   anticipated_rmds: Any
   potential_qcds: str

   icp_data: Any
   formatted_icp_data: str

   models_per_accounts: Any

   beneficiaries_data: Any

def fw_get_account_numbers(
   input: CustodiansGraph
):
   """
   Gets the account numbers for a given client name.
   """
   try:
       model = AzureChatOpenAI(
```

```
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        json_parser = JsonOutputParser(pydantic_object=AccountNumbers)

        household_service = HouseholdLookupService()
        client_name =
household_service.find_household_by_individual(input['client_name'])

        accounts = data_warehouse_fidelity_wealthscape_agent(
            user_full_requirement=f"Get the account numbers for {client_name} and its
types per each client. Just for that/those client name(s), not for other similar
names. However, you are allowed to always include UTMA accounts, even if the names are
different. You must present the account numbers per each client name.",
            is_ui_based=True
        )

        prompt = f"""
        You have been provided with a list of account numbers for a given client.
        Extract only the account numbers and return them in a list.

        Context: {accounts}

        ### You must format your response in the following JSON format:
        {json_parser.get_format_instructions()}
        """

        account_number_list = json_parser.invoke(model.invoke(prompt).content)

        logger.info(f"Accounts: {accounts}")
        logger.info(f"Account Number List: {account_number_list}")

        return {'account_numbers': accounts, 'account_number_list':
account_number_list}
    except Exception as e:
        logger.error(f"Error in fw_get_account_numbers: {e}")
        raise
```

```
class AccountBalances(BaseModel):
   account_number: str = Field(..., description="The account number of the client.")
   account_balance: float = Field(..., description="The account balance of the
client.")

class AccountBalancesPairs(BaseModel):
   pairs: List[AccountBalances] = Field(..., description="The account numbers and
their respective account balances.")

def fw_get_account_balances(
   input: CustodiansGraph
):
   """
   Gets the account balances for a given client name.
   """
   try:
       logger.info(f"Account Number List:
{input['account_number_list']['account_numbers']}")

       account_balances = data_warehouse_fidelity_wealthscape_agent(
           user_full_requirement=f"Get the account balances for
{input['account_number_list']['account_numbers']}. Just for that/those client(s), not
for other similar names. However, you are allowed to always include UTMA accounts,
even if the names are different. Show me the account balances per each account but
only present the Account Value as the Account Balance, organized by account number and
type. You must present the account balances per each client name.",
           is_ui_based=True
       )

       json_parser = JsonOutputParser(pydantic_object=AccountBalancesPairs)

       model = AzureChatOpenAI(
           azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
           api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
           temperature=0,
           max_tokens=None,
           timeout=None,
           max_retries=2
       )

       prompt = f"""
```

```
        Given a list of accounts with their types and account balances, you must only
extract the account numbers and their respective account balances.

        Here is the context:
        {account_balances}

        You must return this as a JSON format:
        {json_parser.get_format_instructions()}
        """

        account_balances_json = json_parser.invoke(model.invoke(prompt).content)

        logger.info(f"Account Balances: {account_balances}")

        logger.info(f"Account Balances JSON: {account_balances_json}")

        total_value = 0.0

        for account_balance in account_balances_json['pairs']:
            total_value += float(account_balance['account_balance'])

        account_balances_json['total_value'] = total_value

        logger.info(f"Total Value: {total_value}")

        return {'account_balances': account_balances, 'account_balances_json':
account_balances_json}
    except Exception as e:
        logger.error(f"Error in fw_get_account_balances: {e}")
        raise

def fw_get_positions(
    input: CustodiansGraph
):
    """
    Gets the positions for a given client name.
    """

    try:
        # household_service = HouseholdLookupService()
        # client_name =
household_service.find_household_by_individual(input['client_name'])
```

```
        # positions = data_warehouse_fidelity_wealthscape_agent(
        #     user_full_requirement=f"""
        #     Get all Treasuries and Certificates of Deposit (CDs) for {client_name}.
        #     """,
        #     is_ui_based=True
        # )

        from
api.app.agents.data_warehouse.tools.data_warehouse_queries.fidelity_functions.position
s import get_fixed_income_positions

        fixed_income_positions =
get_fixed_income_positions(input['account_number_list']['account_numbers'])

        logger.info(f"Fixed Income Positions: {fixed_income_positions}")

        return {'positions': fixed_income_positions}
    except Exception as e:
        logger.error(f"Error in fw_get_positions: {e}")
        raise

def bd_get_transactions_v2(
    account_number: str,
    action: str,
    date_from: str,
    date_to: str,
    cursor=None
):
    """
    Gets the transactions for a given client name.
    """
    try:
        transactions = get_transactions(
            account_number=account_number,
            action=action,
            date_from=date_from,
            date_to=date_to,
            cursor=cursor
        )

        return filter_transactions_basic_bd_v2(transactions)
```

```
   except Exception as e:
       logger.error(f"Error in bd_get_transactions_v2: {e}")
       raise

# def fw_get_transactions(
#     input: CustodiansGraph
# ):
#     """
#     Gets the transactions for a given client name.
#     """
#     try:
#         #household_service = HouseholdLookupService()
#         #client_name =
household_service.find_household_by_individual(input['client_name'])

#         # transaction_types = [
#         #     #"Sell",
#         #     #"Dividend",
#         #     #"Interest",
#         #     "Deposit/Contribution",
#         #     #"Retirement",
#         #     "Withdrawal",
#         #     "Exchange/Transfer"
#         #     #"Other"
#         # ]

#         transaction_types = [
#             "Contribution",
#             "Withdrawal"
#         ]

#         transaction_values = {}

#         logger.info(f"Account Number List:
{input['account_number_list']['account_numbers']}")
#         logger.info(f"Transaction Types: {transaction_types}")

#         for account_number in input['account_number_list']['account_numbers']:

#             transaction_values[account_number] = {}

#             for transaction_type in transaction_types:
```

```
#                 # transaction_temp = get_transactions_core_v2(
#                 #     account_number=account_number,
#                 #     transaction_type=transaction_type,
#                 #     date_from=f"{datetime.now().year}-01-01",
#                 #     date_to=datetime.now().strftime("%Y-%m-%d")
#                 # )

#                 transaction_temp = bd_get_transactions_v2(
#                     account_number=account_number,
#                     action=transaction_type,
#                     date_from=f"{datetime.now().year}-01-01",
#                     date_to=datetime.now().strftime("%Y-%m-%d")
#                 )

#                 if transaction_type == "Contribution":
#                     transaction_type = "contributions"
#                 elif transaction_type == "Withdrawal":
#                     transaction_type = "withdrawals"

#                 transaction_values[account_number][transaction_type] =
transaction_temp

#                 logger.info(f"Transaction for {account_number} and
{transaction_type} - DONE")

#         logger.info(f"Transactions: {transaction_values}")

#         totals_transactions =
get_bd_transactions_for_all_accounts(transaction_values)

#         #totals_transactions =
get_fw_distributions_and_contributions_rate_ytd(input['account_number_list']['account_
numbers'])

#         formatted_transactions = format_transactions(transaction_values,
totals_transactions)

#         return {'transactions': transaction_values, 'formatted_transactions':
formatted_transactions, 'totals_transactions': totals_transactions}
#     except Exception as e:
#         logger.error(f"Error in fw_get_transactions: {e}")
```

```
#         raise

from psycopg2.extras import RealDictCursor

def fw_get_transactions(input: CustodiansGraph):
   try:
       # transaction_types = ["Contribution", "Withdrawal"]
       # transaction_values = {}

       # dsn = MegaTableManager._create_dsn(os.getenv("DW_PGDATABASE_BLACKDIAMOND"))
       # with psycopg2.connect(dsn) as conn:
       #     with conn.cursor(cursor_factory=RealDictCursor) as cursor:

       #         for account_number in
input['account_number_list']['account_numbers']:
       #             transaction_values[account_number] = {}

       #             for transaction_type in transaction_types:
       #                 transaction_temp = bd_get_transactions_v2(
       #                     account_number=account_number,
       #                     action=transaction_type,
       #                     date_from=f"{datetime.now().year}-01-01",
       #                     date_to=datetime.now().strftime("%Y-%m-%d"),
       #                     cursor=cursor
       #                 )

       #                 key = "contributions" if transaction_type == "Contribution"
else "withdrawals"
       #                 transaction_values[account_number][key] = transaction_temp

       # totals_transactions =
get_bd_transactions_for_all_accounts(transaction_values)

       # logger.info(f"Totals Transactions: {totals_transactions}")

       # formatted_transactions = format_transactions(transaction_values,
totals_transactions)

       final_result = dw_transaction_tool_agenda_generator(
           date_from=f"{datetime.now().year}-01-01",
           date_to=datetime.now().strftime("%Y-%m-%d"),
           account_numbers=input['account_number_list']['account_numbers'],
```

```
            provider="fidelity" #TO DO FOR CHARLES SCHWAB
        )

        prior_year_final_result = dw_transaction_tool_agenda_generator(
            date_from=f"{datetime.now().year-1}-01-01",
            date_to=f"{datetime.now().year-1}-12-31",
            account_numbers=input['account_number_list']['account_numbers'],
            provider="fidelity" #TO DO FOR CHARLES SCHWAB
        )

        logger.info(f"Final Result: {final_result}")

        return {
            'transactions': final_result['transactions'],
            'formatted_transactions': final_result['formatted_transactions'],
            'formatted_transactions_prior_year':
prior_year_final_result['formatted_transactions'],
            'totals_transactions': final_result['totals_transactions'],
            'prior_year_total_transactions':
prior_year_final_result['totals_transactions'],
            'totals_transactions_with_tax':
final_result['totals_transactions_with_tax'],
            'prior_year_total_transactions_with_tax':
prior_year_final_result['totals_transactions_with_tax']
        }

    except Exception as e:
        logger.error(f"Error in fw_get_transactions: {e}")
        raise

def dw_transaction_tool(
    client_name: str,
    transaction_type: Annotated[str, "The type of transaction to get. Can be
'Contribution' or 'Withdrawal'"]
):
    """
    Gets the transactions for a given client name.
    """
    try:
        emit_message(f"Getting transactions for {client_name} and {transaction_type}
\n")
```

```
        account_numbers = fw_get_account_numbers({
            "client_name": client_name
        })

        logger.info(f"Account Numbers: {account_numbers}")

        result = fw_get_transactions(
            {
                "account_number_list": account_numbers['account_number_list']
            }
        )

        return {
            "transactions": result['transactions'],
            "totals_transactions": result['totals_transactions']
        }
    except Exception as e:
        logger.error(f"Error in dw_transaction_tool: {e}")
        raise

def get_fw_distributions_and_contributions_rate_ytd(
    account_numbers: List[str]
) -> dict:
    """
    Returns the total YTD withdrawals and contributions across all given accounts,
    along with their respective percentage shares.
    """
    try:
        total_withdrawals = 0.0
        total_contributions = 0.0

        for account_number in account_numbers:
            # Get withdrawals (normalize to positive values)
            withdrawals = get_transactions_core_v2(
                account_number=account_number,
                transaction_type="Withdrawal",
                date_from=f"{datetime.now().year}-01-01",
                date_to=datetime.now().strftime("%Y-%m-%d")
            )
            total_withdrawals += sum(abs(t.get("value", 0.0)) for t in withdrawals)

            # Get contributions (assumed positive already)
```

```
            contributions = get_transactions_core_v2(
                account_number=account_number,
                transaction_type="Deposit/Contribution",
                date_from=f"{datetime.now().year}-01-01",
                date_to=datetime.now().strftime("%Y-%m-%d")
            )
            total_contributions += sum(t.get("value", 0.0) for t in contributions)

        total_flows = total_contributions + total_withdrawals

        if total_flows == 0:
            contributions_rate = 0.0
            withdrawals_rate = 0.0
        else:
            contributions_rate = round((total_contributions / total_flows) * 100, 2)
            withdrawals_rate = round((total_withdrawals / total_flows) * 100, 2)

        return {
            "total_contributions": total_contributions,
            "total_withdrawals": total_withdrawals
        }

    except Exception as e:
        logger.error(f"Error in get_distributions_and_contributions_rate_ytd: {e}")
        raise

def get_bd_distributions_and_contributions_rate_ytd(
    account_number: str,
    transactions: Dict[str, list]
) -> Dict[str, float]:
    """
    Compute YTD withdrawals and contributions for a given account,
    using pre-separated transaction types (withdrawals, distributions).

    Parameters:
        account_number (str): The account to compute values for.
        transactions (dict): {
            "withdrawals": list of withdrawal transactions,
            "distributions": list of contribution transactions
        }

    Returns:
```

```
        dict: {
            "total_contributions": float,
            "total_withdrawals": float,
            "contributions_rate": float,
            "withdrawals_rate": float
        }
    """
    current_year = datetime.now().year
    total_withdrawals = 0.0
    total_contributions = 0.0

    # Process withdrawals
    for txn in transactions.get("withdrawals", []):
        if txn.get("account_number") != account_number:
            continue
        value = float(txn.get("value", 0.0))
        total_withdrawals += abs(value)

    # Process contributions (i.e. distributions)
    for txn in transactions.get("contributions", []):
        if txn.get("account_number") != account_number:
            continue
        value = float(txn.get("value", 0.0))
        total_contributions += value

    total_flows = total_withdrawals + total_contributions

    if total_flows == 0:
        contributions_rate = 0.0
        withdrawals_rate = 0.0
    else:
        contributions_rate = round((total_contributions / total_flows) * 100, 2)
        withdrawals_rate = round((total_withdrawals / total_flows) * 100, 2)

    return {
        str(account_number): {
            "total_contributions": round(total_contributions, 2),
            "total_withdrawals": round(total_withdrawals, 2),
        }
    }

def get_bd_transactions_for_all_accounts(
```

```
    data: Dict[str, Any]
) -> Dict[str, Any]:
    """
    Processes YTD transactions for all accounts and computes global totals.
    """
    try:
        final_result = {}
        total_contributions = 0.0
        total_withdrawals = 0.0

        for account_number, value in data.items():
            account_result = get_bd_distributions_and_contributions_rate_ytd(
                account_number=account_number,
                transactions=value
            )

            final_result[account_number] = account_result

            total_contributions +=
account_result[account_number].get("total_contributions", 0.0)
            total_withdrawals +=
account_result[account_number].get("total_withdrawals", 0.0)

        return {
            "transactions_per_account": final_result,
            "totals": {
                "total_contributions": round(total_contributions, 2),
                "total_withdrawals": round(total_withdrawals, 2)
            }
        }

    except Exception as e:
        print(f"Error in get_bd_transactions_for_all_accounts: {e}")
        raise

class NameAgePair(BaseModel):
    name: str = Field(..., description="The name of the client.")
    age: int = Field(..., description="The age of the client.")

class NameAgePairs(BaseModel):
    name_age_pairs: List[NameAgePair] = Field(..., description="The name and age of the
client(s).")
```

```
def get_all_ages(
    client_name: str
):
    """
    Gets the ages for a given client name.
    """

    try:

        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        client_age = data_warehouse_fidelity_wealthscape_agent(f"""
        How old is/are {client_name}? Return exact numbers, NO DECIMALS. Return all the
ages.
        """)

        logger.info(f"Client Age: {client_age}")

        json_parser = JsonOutputParser(pydantic_object=NameAgePairs)

        client_age_number = (json_parser.parse(
            model.invoke(f"""
            Given this message, {client_age}, return the age(s) as a number and the
names for each client. Return all the ages.
            You should return this as a JSON format:
{json_parser.get_format_instructions()}
            """).content
        ))

        logger.info(f"Client Age Number: {client_age_number}")

        return client_age_number
    except Exception as e:
        logger.error(f"Error in get_all_ages: {e}")
```

```
        raise

def core_anticipated_rmds(
    account_number: float,
    age: int
):
    """
    Anticipated (projected) RMD for clients aged 62 or older but before RMD age (73).
    - Projects the current IRA balance forward to age 73 at a fixed annual growth rate,
      then estimates the first-year RMD using a 4% simplification rule.

    Parameters
    ----------
    account_number : float
        Current IRA balance (USD).
    age : int
        Current client age in years.

    Returns
    -------
    dict | None
        If age < 62, returns None.
        Otherwise returns:
        {
          "age": int,
          "years_to_73": int,
          "growth_rate": float,
          "projected_balance_at_73": float,
          "anticipated_rmd_4pct": float,
          "implied_pct": float
        }
    """
    try:
        # --- configurable constants ---
        GROWTH_RATE = 0.06          # 6% annual growth assumption
        RULE_OF_THUMB_PCT = 0.04    # 4% simplification

        # --- validations ---
        if not isinstance(age, int):
            raise ValueError("age must be an integer")
        if age < 0 or age > 120:
            raise ValueError("age must be between 0 and 120")
```

```
        if account_number is None:
            raise ValueError("account_number (current IRA balance) is required")
        try:
            balance_now = float(account_number)
        except Exception:
            raise ValueError("account_number must be numeric")
        if balance_now < 0:
            raise ValueError("account_number (balance) cannot be negative")

        # Only calculate if age >= 62
        if age < 62:
            return None

        # Ensure ages >= 73 don't project beyond RMD start
        years_to_73 = max(73 - age, 0)

        # --- projection to 73 ---
        projected_balance = balance_now * ((1.0 + GROWTH_RATE) ** years_to_73)

        # --- anticipated RMD at 73 (rule of thumb) ---
        anticipated_rmd_4pct = projected_balance * RULE_OF_THUMB_PCT

        result = {
            "age": age,
            "years_to_73": years_to_73,
            "growth_rate": GROWTH_RATE,
            "projected_balance_at_73": round(projected_balance, 2),
            "anticipated_rmd_4pct": round(anticipated_rmd_4pct, 2),
            "implied_pct": RULE_OF_THUMB_PCT,
        }
        return result

    except Exception as e:
        logger.error(f"Error in core_anticipated_rmds: {e}")
        raise

class NameAccountNumberPair(BaseModel):
    name: str = Field(..., description="The name of the client.")
    account_number: str = Field(..., description="The account number of the client.")
    account_balance: float = Field(..., description="The account balance of the
client.")
```

```
class NameAccountNumberPairs(BaseModel):
   pairs: List[NameAccountNumberPair] = Field(..., description="The name and account
number of the client(s).")

def fw_get_anticipated_rmds(
   input: CustodiansGraph
):
   """
   Gets the anticipated RMDs for a given client name.
   """

   try:

       household_service = HouseholdLookupService()
       client_name =
household_service.find_household_by_individual(input['client_name'])

       name_age_pairs = get_all_ages(
           client_name=client_name
       )

       logger.info(f"Name Age Pairs: {name_age_pairs}")

       json_parser = JsonOutputParser(pydantic_object=NameAccountNumberPairs)

       model = AzureChatOpenAI(
           azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
           api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
           temperature=0,
           max_tokens=None,
           timeout=None,
           max_retries=2
       )

       prompt = f"""
       Given these account balances, extract the client name, account number, and
account balance.
       These must be IRA accounts (Traditional, IRAB, Simple or SEP), not other
account types like ROTH.
       DON'T INCLUDE ANY OTHER ACCOUNT TYPES LIKE ROTH, TODI, 401K, etc. ONLY THE IRA
ACCOUNTS (Traditional, IRAB, Simple or SEP).
```

```
Here is the context:
{input['account_balances']}

Use the same names as these:
{name_age_pairs['name_age_pairs']}

You should return this as a JSON format:
{json_parser.get_format_instructions()}
"""

result = json_parser.parse(
    model.invoke(prompt).content
)

# Group account balances by name
grouped_accounts = {}
for pair in result['pairs']:
    if pair['name'] not in grouped_accounts:
        grouped_accounts[pair['name']] = []
    grouped_accounts[pair['name']].append({
        "account_number": pair['account_number'],
        "account_balance": pair['account_balance']
    })

# Calculate anticipated RMDs for each person
anticipated_rmds = {"rows": []}
for name_age_pair in name_age_pairs['name_age_pairs']:
    name = name_age_pair['name']
    age = name_age_pair['age']

    if name in grouped_accounts:
        for account in grouped_accounts[name]:
            rmd_calc = core_anticipated_rmds(
                account_number=account["account_balance"],
                age=age
            )
            if rmd_calc:  # Only add if RMD calculation returned a result
                anticipated_rmds["rows"].append({
                    "account": f"{account['account_number']} (IRA)",
                    "owner": name,
                    "rmdAmount": rmd_calc['anticipated_rmd_4pct'],
                    "taxImpact": "Advisor Review Required",
```

```
                            "advisorReviewRequired": True,
                            "isAnticipated": True
                        })



        logger.info(f"Anticipated RMDs: {anticipated_rmds}")

        return {'anticipated_rmds': anticipated_rmds}

    except Exception as e:
        logger.error(f"Error in fw_get_anticipated_rmds: {e}")
        raise


def fw_get_rmds(
    input: CustodiansGraph
):
    """
    Gets both current RMDs (age 73+) and anticipated RMDs (age 62-72) for clients.
    """

    try:
        logger.info(f"Input received in fw_get_rmds: {input}")

        household_service = HouseholdLookupService()
        logger.info(f"About to access input['client_name']")
        client_name =
household_service.find_household_by_individual(input['client_name'])
        logger.info(f"Successfully got client_name: {client_name}")

        # Get ages for all clients
        logger.info(f"About to call get_all_ages with client_name: {client_name}")
        name_age_pairs = get_all_ages(client_name=client_name)
        logger.info(f"Name Age Pairs full response: {name_age_pairs}")
        logger.info(f"Type of name_age_pairs: {type(name_age_pairs)}")

        # Get account balances
        logger.info(f"About to access input['account_balances']")
        logger.info(f"Account balances from input: {input.get('account_balances', 'NOT
FOUND')}")
        json_parser = JsonOutputParser(pydantic_object=NameAccountNumberPairs)
```

```
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        # Extract IRA accounts and balances
        balance_prompt = f"""
        Given these account balances, extract the client name, account number, and
account balance.
        These must be IRA accounts (Traditional, IRAB, Simple or SEP), not other
account types like ROTH.
        DON'T INCLUDE ANY OTHER ACCOUNT TYPES LIKE ROTH, TODI, 401K, etc. ONLY THE IRA
ACCOUNTS (Traditional, IRAB, Simple or SEP).

        Here is the context:
        {input['account_balances']}

        Use the same names as these:
        {name_age_pairs['name_age_pairs']}

        You should return this as a JSON format:
        {json_parser.get_format_instructions()}
        """

        account_result = json_parser.parse(model.invoke(balance_prompt).content)

        # Group accounts by name
        logger.info(f"About to process account_result: {account_result}")
        logger.info(f"Type of account_result: {type(account_result)}")

        grouped_accounts = {}
        logger.info(f"About to access account_result['pairs']")
        for pair in account_result['pairs']:
            logger.info(f"Processing pair: {pair}")
            logger.info(f"About to access pair['name']")
            if pair['name'] not in grouped_accounts:
                grouped_accounts[pair['name']] = []
```

```
            grouped_accounts[pair['name']].append({
                "account_number": pair['account_number'],
                "account_balance": pair['account_balance']
            })
        logger.info(f"Finished grouping accounts: {grouped_accounts}")

        # Process RMDs based on age
        current_rmd_clients = []  # age 73+
        anticipated_rmd_results = {"rows": []}  # age 62-72

        logger.info(f"About to process name_age_pairs: {name_age_pairs}")
        logger.info(f"About to access name_age_pairs['name_age_pairs']")
        for name_age_pair in name_age_pairs['name_age_pairs']:
            logger.info(f"Processing name_age_pair: {name_age_pair}")
            name = name_age_pair['name']
            age = name_age_pair['age']
            logger.info(f"Processing client - Name: {name}, Age: {age}")

            if age >= 73:
                logger.info(f"Adding {name} to current_rmd_clients (age >= 73)")
                current_rmd_clients.append(name)
            elif age >= 62:
                if age == 72:
                    logger.info(f"Checking {name} for anticipated RMDs (age == 72)")

                    rmds_for_72 = data_warehouse_fidelity_wealthscape_agent(
                        user_full_requirement=f"Get all RMDs for {name}. Just for that
client, not for other similar names. You must always return the estimated required
minimum distribution amount and the year to date distribution amount."
                    )

                    prompt = f"""
                    You are a financial advisor. You are given an analysis if RMDs
exist or not for a client.
                    You ONLY need to say if RMDs exist or not, no other format is
allowed.
                    You must just return 'yes' or 'no', no other format is allowed.

                    The context for analyzing RMDs existence is here:
                    {rmds_for_72}
                    """
```

```
                    analysis_for_rmds = model.invoke(prompt).content

                    if analysis_for_rmds.lower() == 'yes':
                        current_rmd_clients.append(name)
                    else:
                        logger.info(f"Checking {name} for anticipated RMDs (62 <= age <
73)")
                        if name in grouped_accounts:
                            logger.info(f"Found accounts for {name}:
{grouped_accounts[name]}")
                            for account in grouped_accounts[name]:
                                logger.info(f"Processing account for anticipated RMD
calculation: {account}")
                                logger.info(f"About to call core_anticipated_rmds with
balance: {account['account_balance']} and age: {age}")
                                rmd_calc = core_anticipated_rmds(
                                    account_number=account["account_balance"],
                                    age=age
                                )
                                if rmd_calc:
                                    anticipated_rmd_results["rows"].append({
                                        "account": f"{account['account_number']}
(IRA)",

                                        "owner": name,
                                        "rmdAmount": rmd_calc['anticipated_rmd_4pct'],
                                        "taxImpact": "Advisor Review Required",
                                        "advisorReviewRequired": True,
                                        "isAnticipated": True
                                    })
                else:
                    logger.info(f"Checking {name} for anticipated RMDs (62 <= age <
73)")
                    if name in grouped_accounts:
                        logger.info(f"Found accounts for {name}:
{grouped_accounts[name]}")
                        for account in grouped_accounts[name]:
                            logger.info(f"Processing account for anticipated RMD
calculation: {account}")
                            logger.info(f"About to call core_anticipated_rmds with
balance: {account['account_balance']} and age: {age}")
                            rmd_calc = core_anticipated_rmds(
                                account_number=account["account_balance"],
```

```
                            age=age
                        )
                        if rmd_calc:
                            anticipated_rmd_results["rows"].append({
                                "account": f"{account['account_number']} (IRA)",
                                "owner": name,
                                "rmdAmount": rmd_calc['anticipated_rmd_4pct'],
                                "taxImpact": "Advisor Review Required",
                                "advisorReviewRequired": True,
                                "isAnticipated": True
                            })

    # Get current RMDs if there are any clients 73+
    formatted_rmds = {"rows": []}  # Initialize with empty rows
    rmds = None
    if current_rmd_clients:
        rmds = data_warehouse_fidelity_wealthscape_agent(
            user_full_requirement=f"Get all RMDs for {',
'.join(current_rmd_clients)}. Just for those clients, not for other similar names. The
format of your response must include the ACCOUNT NUMBER, the estimated required
minimum distribution amount and the year to date distribution amount for EACH IRA
account (Traditional, IRAB, Simple or SEP). YOU MUST NOT HALLUCINATE ANY RMDs, ONLY
RETURN THE ACTUAL RMDs FOR THE EXISTENT CLIENTS."
        )

        logger.info(f"Current RMDs: {rmds}")

        if rmds:
            rmd_parser = JsonOutputParser(pydantic_object=RMDsUtil)
            formatted_rmds_prompt = f"""
            You are a financial advisor. You are given RMD data for clients.
            You need to extract and format this data into a structured JSON format.

            CRITICAL RULES:
            - ONLY include IRA accounts (Traditional IRA, IRAB, Simple IRA, or SEP
IRA)
            - DO NOT include ROTH, TODI, 401K, or any other account types
            - Each RMD entry MUST have: account number, estimated required minimum
distribution amount, and year to date distribution amount
            - Extract ALL available RMD data from the text below
            - If the account number is mentioned anywhere in the text (e.g., "for
Donna Osipchak", "Account #12345", etc.), include it
```

```
                - If no valid IRA RMDs are found, return {{"rows": []}}

                RMD Data to parse:
                {rmds}

                IMPORTANT: Look carefully at the text above. Extract every piece of RMD
information you can find, including:
                - Client names (use as accountNumber if no account number is provided)
                - Account numbers
                - Estimated Required Minimum Distribution Amount
                - Year to Date Distribution Amount

                YOU MUST RETURN ONLY THE JSON OBJECT, NO OTHER TEXT.

                Format your response exactly as:
                {rmd_parser.get_format_instructions()}
                """

                raw_result = model.invoke(formatted_rmds_prompt).content

                logger.info(f"Raw result: {raw_result}")

                formatted_rmds = rmd_parser.parse(raw_result)

                logger.info(f"About to process formatted_rmds: {formatted_rmds}")

                temp_var = 0

                if not formatted_rmds.get('rows'):
                    return {
                        'rmds': [],
                        'formatted_rmds': {'rows': []},
                        'anticipated_rmds': {'rows': []}
                    }

                for i in range(len(formatted_rmds['rows'])):

                    # Convert amounts to float, removing $ if present
                    rmd_amount =
float(str(formatted_rmds['rows'][i]['rmdAmount']).replace('$', '').replace(',', ''))
```

```
                    ytd_amount =
float(str(formatted_rmds['rows'][i]['yearToDateDistributionAmount']).replace('$',
'').replace(',', ''))

                    # Calculate and add leftToSatisfy field
                    formatted_rmds['rows'][i]['leftToSatisfy'] = f"${max(0, rmd_amount
- ytd_amount):,.2f} (${ytd_amount if ytd_amount < rmd_amount else ytd_amount} sent out
YTD)"

                    formatted_rmds['rows'][i]['taxImpact'] = 'Advisor Review Required'
                    formatted_rmds['rows'][i]['advisorReviewRequired'] = True
                    formatted_rmds['rows'][i]['isAnticipated'] = False

                logger.info(f"Formatted Current RMDs: {formatted_rmds}")

        logger.info(f"Anticipated RMDs: {anticipated_rmd_results}")

        return {
            'rmds': rmds,
            'formatted_rmds': formatted_rmds,
            'anticipated_rmds': anticipated_rmd_results
        }
    except Exception as e:
        logger.error(f"Error in fw_get_rmds: {e}")
        raise

def fw_get_potential_qcds(
    input: CustodiansGraph
):
    """
    Gets the potential QCDs for a given client name.
    """

    try:
        household_service = HouseholdLookupService()
        client_name =
household_service.find_household_by_individual(input['client_name'])

        potential_qcds = data_warehouse_fidelity_wealthscape_agent(
            user_full_requirement=f"Get all QCDs for {client_name}. Just for that
client, not for other similar names. If there are no confirmed QCDs, you must return
that None Exist, don't force to share results that are not valid.",
```

```
            is_ui_based=True
        )

        logger.info(f"Potential QCDs: {potential_qcds}")

        return {'potential_qcds': potential_qcds}
    except Exception as e:
        logger.error(f"Error in fw_get_potential_qcds: {e}")
        raise

def fw_get_icp_data(
    input: CustodiansGraph
):
    """
    Gets the ICP data for a given client name.
    """
    try:
        household_service = HouseholdLookupService()
        client_name =
household_service.find_household_by_individual(input['client_name'])

        icp_data = {}

        for account_number in input['account_number_list']['account_numbers']:

            icp_data[account_number] = {}

            logger.info(f"Getting ICP Data for account {account_number}")
            logger.info(f"Date from: {datetime.now().year}-01-01")
            logger.info(f"Date to: {datetime.now().strftime('%Y-%m-%d')}")

            icp_temp = get_icp_traces_v2(
                account_number=account_number,
                date_from=f"{datetime.now().year}-01-01",
                date_to=datetime.now().strftime("%Y-%m-%d")
            )

            icp_data[account_number] = icp_temp

            logger.info(f"ICP Data for account {account_number} - DONE")

        logger.info(f"ICP Data: {icp_data}")
```

```
        formatted_icp_data = format_cash_flow_data(icp_data)

        return {'icp_data': icp_data, 'formatted_icp_data': formatted_icp_data}
    except Exception as e:
        logger.error(f"Error in fw_get_positions: {e}")
        raise

def fw_get_additional_data(
    input: CustodiansGraph
):
    """
    Gets the additional data for a given client name.
    """
    pass

def get_beneficiaries_data(
    input: CustodiansGraph
):
    """
    Gets the beneficiaries data for a given client name.
    """

    try:
        household_service = HouseholdLookupService()
        client_name =
household_service.find_household_by_individual(input['client_name'])

        orm_instance_advisor = LookupClientAdvisorService()
        advisor_record =
orm_instance_advisor.get_latest_record(client_name=input['client_name'])

        advisor_name = advisor_record['advisor_name']

        sharepoint_db_manager =
AzurePostgreSQLManager(table_name="beneficiaries_vector_store")

        response = sharepoint_db_manager.read_latest_by_client_and_advisor(
            client_name=client_name,
            advisor_name=advisor_name
        )
```

```
        content = response.get("content") if response else ""

        logger.info(f"Beneficiaries Data: {content}")

        return {'beneficiaries_data': content}
    except Exception as e:
        logger.error(f"Error in get_beneficiaries_data: {e}")
        raise

def format_model_per_accounts(data_list):
                formatted = []

                for item in data_list:
                    account_number = item.get("account_number", "Unknown")
                    comparisons = item.get("comparisons", [])

                    if comparisons:  # caso con modelo asignado
                        for comp in comparisons:
                            target_name = comp.get("target_name", "Unknown")
                            formatted.append(f"Account: {account_number} | Target:
{target_name}")
                    else:  # caso sin modelo asignado
                        formatted.append(f"Account: {account_number} | Target: None (No
model assigned)")

                return formatted

def fw_get_model_per_accounts(
    input: CustodiansGraph
):
    """
    Gets the target vs holdings comparison for a given client name.
    """

    try:
        account_numbers = input['account_number_list']['account_numbers']

        model_per_accounts_data = get_strategies(account_numbers=account_numbers)

        logger.info(f"Model per Accounts Data: {model_per_accounts_data}")

        #final_result = format_model_per_accounts(model_per_accounts_data)
```

```
        logger.info(f"Final Result: {model_per_accounts_data}")

        return {'models_per_accounts': model_per_accounts_data}
    except Exception as e:
        logger.error(f"Error in fw_get_model_per_accounts: {e}")
        raise

def merge_custodian_node(state: CustodiansGraph):
    return state

def build_custodians_graph(custodian_id: int, applicability_to_client: dict,
client_age_number: int):
    """
    Builds a graph for a given client name.
    """

    # Create the Graph
    graph_builder = StateGraph(CustodiansGraph)

    graph_builder.add_node("merge_custodian_node", merge_custodian_node, defer=True)

    if custodian_id == 1:
        graph_builder.add_node("fw_get_account_numbers_node", fw_get_account_numbers)
        graph_builder.add_node("fw_get_account_balances_node", fw_get_account_balances)
        graph_builder.add_node("fw_get_positions_node", fw_get_positions)
        graph_builder.add_node("fw_get_rmds_node", fw_get_rmds)
        graph_builder.add_node("fw_get_potential_qcds_node", fw_get_potential_qcds)
        graph_builder.add_node("fw_get_transactions_node", fw_get_transactions)
        graph_builder.add_node("get_beneficiaries_data_node", get_beneficiaries_data)
        #graph_builder.add_node("fw_get_icp_data_node", fw_get_icp_data)
        graph_builder.add_node("fw_get_anticipated_rmds_node", fw_get_anticipated_rmds)
        graph_builder.add_node("fw_get_model_per_accounts_node",
fw_get_model_per_accounts)

        graph_builder.add_edge(START, "fw_get_account_numbers_node")
        graph_builder.add_edge(START, "get_beneficiaries_data_node")
        graph_builder.add_edge("fw_get_account_numbers_node",
"fw_get_account_balances_node")
        graph_builder.add_edge("fw_get_account_numbers_node",
"fw_get_model_per_accounts_node")
        graph_builder.add_edge("fw_get_account_numbers_node", "fw_get_positions_node")
```

```
        #graph_builder.add_edge("fw_get_account_balances_node", "fw_get_icp_data_node")
        graph_builder.add_edge("fw_get_positions_node", "fw_get_transactions_node")

        if client_age_number >= 62 and client_age_number < 73:
            graph_builder.add_edge("fw_get_account_balances_node",
"fw_get_anticipated_rmds_node")
            graph_builder.add_edge("fw_get_anticipated_rmds_node",
"merge_custodian_node")

        if applicability_to_client['applicable_for_rmds'] and client_age_number >= 73:
            graph_builder.add_edge("fw_get_account_balances_node", "fw_get_rmds_node")
            graph_builder.add_edge("fw_get_rmds_node", "merge_custodian_node")

        if applicability_to_client['applicable_for_qcds']:
            graph_builder.add_edge("fw_get_account_numbers_node",
"fw_get_potential_qcds_node")
            graph_builder.add_edge("fw_get_potential_qcds_node",
"merge_custodian_node")

        #graph_builder.add_edge("fw_get_icp_data_node", END)
        graph_builder.add_edge("get_beneficiaries_data_node", "merge_custodian_node")
        graph_builder.add_edge("fw_get_account_balances_node", "merge_custodian_node")
        graph_builder.add_edge("fw_get_model_per_accounts_node",
"merge_custodian_node")
        graph_builder.add_edge("fw_get_transactions_node", "merge_custodian_node")

        graph_builder.add_edge("merge_custodian_node", END)

    # Compile the Graph
    graph = graph_builder.compile()

    return graph
```

```
from datetime import datetime
import os
from typing import Any, TypedDict

from langchain_openai import AzureChatOpenAI

from api.app.config.logger import logger
from langgraph.graph import START, END, StateGraph
```

```python
class PortfolioAndRiskReviewGraph(TypedDict):
    client_name: str

    black_diamond_context: Any
    nitrogen_context: Any

    client_profile: Any
    employment_and_meeting_insights: Any
    client_applicability: Any

    black_diamond_portfolio_values: str
    black_diamond_portfolio_values_analysis: str
    nitrogen_risk_analysis_values_analysis: str

def review_black_diamond_portfolio_values(
    input: PortfolioAndRiskReviewGraph
):
    """
    This function is used to review the black diamond portfolio values.
    """

    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        today = datetime.now().strftime("%Y-%m-%d")

        prompt = f"""
        You are a financial planning assistant.
        Your task is to extract detailed, structured data from a **Black Diamond
portfolio context**, without assumptions or hallucinations.
        Only report what is explicitly shown — **pull exact numbers** when available.

        Today's date is **{today}**.

        ---
```

```
### 🎯 OBJECTIVE

Extract and organize the following categories from the Black Diamond context.
If a data point is missing or not visible, return:

→ `Not available in Black Diamond. Advisor review recommended.`

Each extracted value must include **Chain of Thought (CoT) reasoning**
explaining:
- Where it was located (e.g., "Performance Summary table", "Pie chart",
"Account Holdings section")
- Why it is included
- Whether it's complete or needs advisor follow-up

---

## 📊 Section 1: Portfolio Performance

- **Quarter-to-Date (QTD) Return (%):**
- **Year-to-Date (YTD) Return (%):**
- **Return Since Benchmark Date (e.g., 1/1/2024):**
- **Account-Level Returns:**
- Account Name / Type
- % Return
- **Performance vs Benchmark:**
- Benchmark name
- Portfolio % vs Benchmark % (if comparative shown)
- **Recent Volatility Details:**
- Period of fluctuation
- % Drop or Gain (if shown)
> Reasoning:

---

## 🧾 Section 2: Account Aggregation

- **Complete List of Linked Accounts:**
- Account name, number (if shown), and type (IRA, Roth, Individual, Trust, HSA,
etc.)
- **Ownership Structure:**
- Joint, individual, trust-owned, inherited, etc.
```

```
      - **Custodian or External Source:**
      - e.g., Fidelity, Schwab, Edward Jones, 401(k) provider
      > Reasoning:

      ---

      ## 📦 Section 3: Holdings Snapshot

      - **Top Holdings by Dollar Value:**
      - Name, $ Value, and % Allocation
      - **Full Allocation Table (if shown). THIS MUST INCLUDE ALL THE EXISTENT
PERCENTAGES FROM THE CURRENT PORTFOLIO ALLOCATION. DO NOT OMIT ANY PERCENTAGE:**
      - Ticker/Fund/Security Name
      - % of Total Portfolio
      - **Sector Exposure Breakdown:**
      - e.g., Tech: 24%, Energy: 9%
      - **Geographic Exposure:**
      - Domestic vs International (with %), Emerging Markets (if labeled)
      > Reasoning:

      ---

      ## 🧮 Section 4: Asset Allocation

      - **By Asset Class:**
      - Equities: __%
      - Bonds (Corporate/Government): __%
      - Alternatives: __%
      - Cash: __%
      - **Deviation from Target Allocation (if shown):**
      - e.g., "Current equities 70%, target 60% → 10% overweight"
      - **Tax Location Efficiency (if indicated):**
      - e.g., Taxable: 60%, Roth: 25%, Traditional IRA: 15%
      > Reasoning:

      ---

      ## 💰 Section 5: Capital Gains Exposure

      - **Embedded Gains in Mutual Funds:**
      - $ Amount or % Unrealized (if shown)
      - **Unrealized Gains or Losses:**
```

```
- Total $ Value per account (taxable only)
- **Harvesting Opportunity Flagged?**
- Description + dollar threshold (if applicable)
> Reasoning:

---

## 💸 Section 6: Fees & Expenses

- **Internal Expense Ratios for Key Holdings:**
- Fund/Security name + % ER
- **Overall Advisory or Management Fees (if shown):**
- e.g., 1.00%, 0.75%, etc.
- **Cost Efficiency Concerns:**
- Highlight expensive funds in taxable or overlaps
> Reasoning:

## Section 7: Risk Profile:
- **Risk Profile:**
- Risk Profile: Conservative, Moderate, Moderate Aggressive, Aggressive
> Reasoning:

---

## 🧠 Advisor-Level Planning Flags (Based on Extracted Data)

Using the extracted values above, flag only if supported by the data:

- **⚠️ High Allocation to Cash:**
- % in cash vs typical target
→ Flag: `"Consider reinvestment discussion if purpose unclear."`

- **⚠️ Portfolio Lag vs Market Benchmark:**
- Portfolio YTD vs S&P 500 or 60/40 benchmark
→ Flag: `"Review investment thesis or rebalance if underperforming."`

- **⚠️ Holding Overlap or Redundancy:**
- Duplicate funds/stocks across accounts
→ Flag: `"Simplification opportunity – reduce redundancy."`

- **⚠️ Concentration Risk:**
- Holding > 10% of portfolio
```

```
        → Flag: `"Discuss diversification for concentrated positions."`

        - **⚠️ Tax Gain/Loss Management Needed:**
        - Unrealized gain/loss > $X
        → Flag: `"Coordinate with tax strategy or harvesting windows."`

        > Justify each flag using exact values from above.

        ⚠️ If a section lacks required data:
        → `"No action triggered - data not present in Black Diamond."`

        ---

        📥 Input Variables:

        - `client_profile`:
        {input['client_profile']}

        - `client_applicability`:
        {input['client_applicability']}

        - `black_diamond_context`:
        {input['black_diamond_context']}
        """

        response = model.invoke(prompt).content

        logger.info(f"Black Diamond Portfolio Values: {response}")

        return {'black_diamond_portfolio_values': response}
    except Exception as e:
        logger.error(f"Error in review_black_diamond_portfolio_values: {e}")
        raise

def analyze_black_diamond_portfolio_values(
    input: PortfolioAndRiskReviewGraph
):
    """
    This function is used to analyze the black diamond portfolio values.
    """

    try:
```

```
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        today = datetime.now().strftime("%Y-%m-%d")

        prompt = f"""
        You are a financial planning AI assistant.
        Your task is to analyze **Black Diamond portfolio data** and surface
**advisor-level planning flags**, based solely on what is explicitly documented in the
data.

        📅 Today's date: **{today}**

        ---

        ### 🎯 OBJECTIVE

        Review the Black Diamond data and identify any **portfolio-related risks,
shifts, or opportunities** that require advisor attention.
        Only trigger a flag if:

        - The data is **clearly present**
        - The insight can be supported with **numeric values or visual elements**
        - It relates directly to investment management, allocation drift, or planning
strategy

        Do **NOT** hallucinate, estimate, or assume unstated values.

        Each triggered flag must include:

        - **Flag Title / Description**
        - **Trigger Logic Explanation (Chain of Thought)**
        - **Exact Data Reference**
        - **Advisor Action Recommendation**

        ---
```

```
### 🚨 Categories of Advisor-Level Flags

#### 1. 📉 Underperformance vs Benchmarks
- Trigger if: Portfolio return (QTD, YTD, 1Y, 3Y, etc.) **lags benchmark** by 5% or more, or shows recurring underperformance.
- Extract:
    - QTD, YTD, and multi-year % returns
    - Benchmark return values (e.g., S&P 500, blended index)
> CoT Reasoning: Use the performance comparison section or tables. State the delta between portfolio and benchmark. Flag whether the underperformance is recent or systemic.
> Advisor Action: “Review performance drag — identify asset class or security lagging expectations.”

---

#### 2. ⚖️ Asset Allocation Drift / Position Shifts
- Trigger if: Allocation has deviated >5–10% from stated model, OR large shift to cash/bonds/equities is visible.
- Extract:
    - % of Equities, Bonds (Gov/Corp), Alternatives, Cash
    - Target vs current drift (if measurable)
> CoT Reasoning: Use asset allocation pie chart or table. Call out classes with increased or reduced exposure.
> Advisor Action: “Discuss realignment or confirm intentional shift. Assess if risk profile is still appropriate.”

---

#### 3. 💰 Capital Gains Exposure or Tax Inefficiency
- Trigger if:
    - Unrealized gains/losses > $10,000 in taxable accounts
    - Mutual funds with large embedded gains in non-qualified accounts
- Extract:
    - Specific $ values of gains/losses by position
    - Fund names and tax status
> CoT Reasoning: Use taxable account holding tables or gain/loss report. Flag only if it impacts planning.
> Advisor Action: “Consider harvesting, repositioning, or using losses to offset gains. Review tax impact.”
```

```
---

#### 4. 🧱 Concentration Risk / Overweight Holdings
- Trigger if:
    - A single holding exceeds 15–20% of total portfolio value
    - Multiple positions in the same sector or issuer
- Extract:
    - Holding name
    - $ value
    - % of portfolio
> CoT Reasoning: From Top Holdings table. Emphasize if this risk spans multiple accounts or lacks diversification.
> Advisor Action: “Discuss diversification options or hedging. Review comfort with concentration.”

---

#### 5. 🌐 External Account Aggregation Shift
- Trigger if:
    - External custodians are listed (Edward Jones, Robinhood, 401(k), etc.)
    - A **new account appears**, a **major holding disappears**, or **balances shift**
- Extract:
    - Custodian/account name
    - Previous vs current holding/balance (if visible)
    - Ownership or registration change (e.g., joint to trust)
> CoT Reasoning: Use account summary or aggregation list. Look for additions/removals or large balance changes.
> Advisor Action: “Confirm asset movement. Integrate into updated plan or verify alignment with funding strategies.”

---

#### 6. 🧠 Additional Triggers (Based on Context)
- High cash allocation (e.g., >20%) → `"Flag reinvestment or intentional hold."`
- Overlap in holdings across accounts → `"Flag for simplification or tax coordination."`
- Expense ratios >1.00% → `"Review fee efficiency of funds."`

---
```

```
        ⚠️ If a section lacks required data:
        → `"No action triggered - data not present in Black Diamond."`

        ---

        ### 📥 Input Variables

        - `client_profile`:
        {input['client_profile']}

        - `client_applicability`:
        {input['client_applicability']}

        - `employment_and_meeting_insights`:
        {input['employment_and_meeting_insights']}

        - `black_diamond_portfolio_values`:
        {input['black_diamond_portfolio_values']}
        """

        response = model.invoke(prompt).content

        logger.info(f"Black Diamond Portfolio Values Analysis: {response}")

        return {'black_diamond_portfolio_values_analysis': response}
    except Exception as e:
        logger.error(f"Error in analyze_black_diamond_portfolio_values: {e}")
        raise

def nitrogen_risk_analysis_values(
    input: PortfolioAndRiskReviewGraph
):
    """
    This function is used to analyze the nitrogen risk analysis values.
    """

    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
```

```
            timeout=None,
            max_retries=2
        )

        today = datetime.now().strftime("%Y-%m-%d")

        prompt = f"""
        You are a financial planning AI assistant.

        Your task is to analyze **Nitrogen risk analysis data** in the context of the client's current portfolio and any recent volatility or life event, using information from **Nitrogen**, **Black Diamond**, and the client's profile or meeting context.

        📅 Today's date: **{today}**

        ---

        ### 🎯 OBJECTIVE

        Review the Nitrogen risk score and determine whether the client's current portfolio matches their risk tolerance.
        Use **Black Diamond performance and allocation data** to validate the current alignment.
        If recent volatility or a life change is detected, **recommend a re-evaluation of risk profile**.

        ---

        YOU MUST NOT SHOW ANY PORTFOLIO ALLOCATION VALUES, JUST RISK SCORES AND FLAGS, NO ALLOCATION/PORTFOLIO VALUES.
        IF YOU REFER TO THE PORTFOLIO, YOU MUST USE THE VALUES FROM "BLACK DIAMOND" AND NOT "RIGHTCAPITAL" NOR "NITROGEN".

        ### 🔍 Required Analysis

        #### 1. 🎯 Risk Score Alignment
        - **Nitrogen Risk Score** (if provided):
        - **Portfolio Risk Score or Modeled Range** (from Nitrogen):
        - **Alignment Check**:
            - Flag if portfolio is **more aggressive or more conservative** than Nitrogen's modeled range
        > CoT Reasoning:
```

```
            - Compare Nitrogen's risk number vs actual allocation in Black Diamond.
            - Flag if there's >10-15% deviation between expected and actual allocation.

        ---

        #### 2. 📉 Recent Volatility (from Black Diamond)
        - Flag if:
            - Portfolio experienced a negative return in any recent period (QTD/YTD),
or
            - Asset class shifts were triggered (e.g., to/from cash)
        - Include:
            - QTD/YTD returns
            - Asset class breakdown
        > CoT Reasoning:
            - Use Black Diamond performance history and allocation tables.
            - Link volatility or shift to potential client discomfort or behavior
change.

        ---

        #### 3. 🔄 Life Event Risk Reevaluation Trigger
        - Flag if:
            - Client profile or recent meeting notes indicate a major life event:
                - Retirement
                - Job change
                - Health change
                - Family event (marriage, divorce, inheritance, etc.)
        - Extract:
            - Any such note from employment_and_meeting_insights or client_profile
        > CoT Reasoning:
            - Life transitions often alter financial priorities or risk appetite.
            - Recommend a risk reassessment if recent changes are found.

        ---

        ### 📌 Final Output Structure

        For each flag, include:
        - **Flag Title**
        - **Trigger Explanation**
        - **Supporting Values**
        - **Advisor Recommendation**
```

```
        ---

        ⚠️ If no data is found in Nitrogen or Black Diamond:
        → `"No risk-related action triggered - review data availability."`

        ---

        ### 📥 Input Variables

        - `client_profile`:
        {input['client_profile']}

        - `client_applicability`:
        {input['client_applicability']}

        - `employment_and_meeting_insights`:
        {input['employment_and_meeting_insights']}

        - `nitrogen_context`:
        {input['nitrogen_context']}

        - `black_diamond_portfolio_values`:
        {input['black_diamond_portfolio_values']}
        """

        response = model.invoke(prompt).content

        logger.info(f"Nitrogen Risk Analysis Values: {response}")

        return {'nitrogen_risk_analysis_values_analysis': response}
    except Exception as e:
        logger.error(f"Error in nitrogen_risk_analysis_values: {e}")
        raise

def portfolio_and_risk_review_graph(nitrogen_context: Any):
    """
    Builds a graph for a given client name.
    """

    # Create the Graph
    graph_builder = StateGraph(PortfolioAndRiskReviewGraph)
```

```
   graph_builder.add_node("review_black_diamond_portfolio_values_node",
review_black_diamond_portfolio_values)
   graph_builder.add_node("analyze_black_diamond_portfolio_values_node",
analyze_black_diamond_portfolio_values)
   graph_builder.add_node("nitrogen_risk_analysis_values_node",
nitrogen_risk_analysis_values)

   graph_builder.add_edge(START, "review_black_diamond_portfolio_values_node")
   graph_builder.add_edge("review_black_diamond_portfolio_values_node",
"analyze_black_diamond_portfolio_values_node")
   graph_builder.add_edge("analyze_black_diamond_portfolio_values_node", END)

   if nitrogen_context:
       graph_builder.add_edge("analyze_black_diamond_portfolio_values_node",
"nitrogen_risk_analysis_values_node")
       graph_builder.add_edge("nitrogen_risk_analysis_values_node", END)

   portfolio_and_risk_review_graph = graph_builder.compile()

   return portfolio_and_risk_review_graph
```

```
from datetime import datetime
import os
from typing import Any, TypedDict
```

```
from langchain_openai import AzureChatOpenAI
```

```
from api.app.config.logger import logger
from api.app.helpers.azure_postgresql_manager import HouseholdLookupService
```

```
from langgraph.graph import StateGraph, END, START
```

```
class PlanningAndRetirementTriggerGraph(TypedDict):
   client_name: str
```

```
   client_profile: Any
```

```
   client_applicability: Any
   rightcapital_context: Any
   retire_up_context: Any

   rightcapital_planning_data: str
   rightcapital_planning_analysis: str
```

```
    retireup_planning_analysis: str

def rightcapital_planning_data(
    input: PlanningAndRetirementTriggerGraph
):
    """
    This function is used to generate the planning data for the rightcapital project.
    """

    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        today = datetime.now().strftime("%Y-%m-%d")

        prompt = f"""
        You are an AI assistant helping a financial advisor prepare for a client's
planning meeting.

        Your task is to extract structured data **only** from RightCapital planning
outputs, and include ONLY the sections that are applicable to the client based on
their profile and eligibility status.

        Today's date is **{today}**.

        ---

        ⚠️ You MUST NOT:
        - Assume any value not explicitly present in the RightCapital context.
        - Perform any calculation, interpretation, or rounding.
        - Project outcomes based on age, eligibility, or inferred logic.
        - Say "Yes" or "No" to strategy modeling unless the context confirms it AND
shows at least partial numeric detail.

        ---

        ### 🔍 Source: RightCapital (Planning Software)

        Use only the information explicitly documented in the RightCapital output.
        If a section or value is missing, return:

        → `Not available in RightCapital. Advisor review recommended.`

        Each field must include **Chain of Thought (CoT) reasoning** explaining:
        - Where the data came from (e.g., Goals tab, Tax tab, Stress Test module)
        - Why it was included
```

```
        - Whether the data is complete or if a review is required

        ---

        ### 🧩 Output Sections - RightCapital Data (Eligibility-Aware)

        ---

        #### 1. 📘 Profile Blueprint
        - **Client Name**:
        - **Primary Retirement Age Target**:
        - **Employment Status (if shown)**:
        - **Expected Retirement Year**:
        - **Household Net Worth (if tracked)**:
        > Reasoning:

        ---

        #### 2. 🎯 Financial Plan (Goals Section)
        - **Major Goals Listed** (e.g., gifting, travel, home purchase):
        - **Target Dates** (for each goal):
        - **Funding Sources Assigned**:
        > Reasoning:

        ---

        #### 3. 🧓 Retirement Planning
        - **Stress Test Modeled?**
        - If Yes: Include projected success %, Monte Carlo score, or numeric result
ranges.
        - **Roth Conversion Strategy Modeled?**
        - If Yes: Include tax year(s) targeted, amount, and projected benefit if shown.

        % if input['client_applicability']['applicable_for_social_security_benefits']:
        - **Social Security Strategy Included?**
        - If Yes: Include earliest/optimal claim age, monthly income projection, and
assumptions noted.
        > Reasoning:
        % endif

        % if input['client_applicability']['applicable_for_medicare_benefits']:
        - **Medicare Assumptions Shown?**
        - If Yes: Summarize modeled Medicare cost assumptions or premium levels if
visible.
        > Reasoning:
        % endif

        ---

        #### 4. 🛡️ Insurance Overview
        - **Active Life Insurance Policies**:
            - Type (e.g., Term, Whole, Universal)
```

```
          - Coverage Amount
          - Carrier Name (if listed)
      > Reasoning:

      ---

      #### 5. 🧾 Tax Planning View
      - **Tax Estimate for Current Year** (dollar amount or bracket):
      - **Tax Strategies Modeled** (e.g., Roth conversions, bracket smoothing,
capital gains plans):
          - Include simulation year(s), trigger condition(s), and outcomes modeled
      > Reasoning:

      ---

      % if input['client_applicability']['applicable_for_rmds'] or
input['client_applicability']['applicable_for_qcds']:
      #### 6. 📤 RMD / QCD Review

      % if input['client_applicability']['applicable_for_rmds']:
      - **RMD Projection or Modeling Details**:
          - First RMD Year?
          - Modeled Withdrawal Amount?
          - Tax impact (if modeled)?
      > Reasoning:
      % endif

      % if input['client_applicability']['applicable_for_qcds']:
      - **QCD Strategy Included?**
      - If Yes: Include expected giving amount, IRA source, and timing if modeled
      > Reasoning:
      % endif

      ---
      % endif

      #### 7. 🏛️ Estate Planning
      - **Checklist Completed?** (Yes/No):
      - **Beneficiary Data Present?**
          - List of account types (IRA, Roth, Trust)
          - Beneficiary names and relationships (if shown)
      > Reasoning:

      ---

      ⚠️ For any section with no data found:
      → `Not available in RightCapital. Advisor review recommended.`
      > Reasoning: "This data point was not shown in the provided RightCapital
context. Review is required to confirm status."

      ---
```

```
        ### 📥 Variables (Required Context)

        - `client_profile`:
        {input['client_profile']}

        - `client_applicability`:
        {input['client_applicability']}

        - `rightcapital_context`:
        {input['rightcapital_context']}
        """

        result = model.invoke(prompt).content

        logger.info(f"RightCapital Planning Data: {result}")

        return {'rightcapital_planning_data': result}
    except Exception as e:
        logger.error(f"Error in rightcapital_planning_data: {e}")
        raise

def rightcapital_planning_analysis(
    input: PlanningAndRetirementTriggerGraph
):
    """
    This function is used to analyze the planning data for the rightcapital project.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        today = datetime.now().strftime("%Y-%m-%d")

        prompt = f"""
        You are a financial planning AI assistant.
        Your task is to analyze RightCapital data to detect **advisor-level planning
flags**, based strictly on the data provided and the client's profile/applicability.

        Today’s date is **{today}**.

        ⚠️ You MUST NOT:
        - Assume any value not shown in the RightCapital context.
        - Perform any calculation or interpretation beyond what is explicitly
documented.
        - Suggest values or outcomes based on client age or logic unless shown in the
input.
```

```
---

### 🎯 OBJECTIVE

From the RightCapital output, extract and list only the **advisor-relevant flags** that require action, follow-up, or review.
You must:

1. Use logic based on the client's **age and applicability**:
    - For example, only flag Social Security if the client is eligible
2. ONLY trigger a flag if:
    - There is clear data modeled or documented in RightCapital
    - OR the data suggests a gap or decision point
3. If a strategy has already been modeled (e.g., Roth conversion), return it as:
    → `"Modeled previously – Advisor check-in recommended"`

---

### 🚨 Advisor Flag Categories to Analyze

Each category must include:

- **Flag Description**
- **Trigger Logic Explanation (CoT)**
- **Advisor Action Recommended**

---

#### 1. 💵 Upcoming Cash Flow Gap
- Trigger if: Retirement drawdowns show a shortfall or unstated income gap
> Reasoning: Check projections for future years with negative or unassigned cash flow.

---

#### 2. 🔁 Roth Conversion Opportunity
- Trigger if:
    - The client is eligible for conversions (based on age & tax bracket)
    - Roth strategy was modeled but not yet implemented
> Reasoning: Look for modeled scenarios. If found → flag as a check-in.

---

#### 3. 📉 Large Year-over-Year Tax Variance
- Trigger if: Tax estimates swing significantly between years (>25%) or indicate a distribution spike
> Reasoning: Compare current year vs future projections.

---
```

```
#### 4. 👴 Pension / Social Security / Medicare Triggers

% if input['client_applicability']['applicable_for_social_security_benefits']:
- **Social Security Strategy Triggered?**
    > Reasoning: Only if modeled. Flag if not implemented yet or nearing suggested start year.
% endif

% if input['client_applicability']['applicable_for_medicare_benefits']:
- **Medicare Enrollment Window?**
    > Reasoning: Flag if client is within 3 months of turning 65 and planning does not yet reflect enrollment.
% endif

% if input['client_applicability']['applicable_for_rmds']:
- **RMD Planning Needed?**
    > Reasoning: Check if first RMD year is upcoming and plan lacks explicit withdrawal modeling.
% endif

% if input['client_applicability']['applicable_for_qcds']:
- **QCD Opportunity?**
    > Reasoning: Flag if client is age 70½+ and plan includes charitable goals but QCD not shown.
% endif

---

#### 5. 💰 Lump Sum or Drawdown Events Modeled
- If a major withdrawal or conversion plan was modeled:
    → Flag as `"Previously modeled. Advisor check-in recommended for implementation review."`
> Reasoning: Use to prompt follow-up and assess status.

---

⚠️ For any category where no data is found:
→ `"No action needed based on current RightCapital data."`

---

### 📥 Input Variables

- `client_profile`:
{input['client_profile']}

- `client_applicability`:
{input['client_applicability']}

- `rightcapital_context`:
{input['rightcapital_planning_data']}
```

```
        """

        result = model.invoke(prompt).content

        logger.info(f"RightCapital Planning Analysis: {result}")

        return {'rightcapital_planning_analysis': result}
    except Exception as e:
        logger.error(f"Error in rightcapital_planning_analysis: {e}")
        raise

def retire_up_planning_analysis(
    input: PlanningAndRetirementTriggerGraph
):
    """
    This function is used to analyze the planning data for the retireup project.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
            max_retries=2
        )

        today = datetime.now().strftime("%Y-%m-%d")

        prompt = f"""
        You are a financial planning AI assistant.
        Your task is to analyze RetireUp data to detect **advisor-level planning
flags**, using only the data exactly as presented.

        ⚠️ You MUST NOT:
        - Assume any value not shown in the RetireUp context.
        - Perform any calculation or interpretation beyond what is explicitly
documented.
        - Suggest values or outcomes based on client age or logic unless shown in the
input.

        Today’s date is **{today}**.

        ---

        ### 🎯 OBJECTIVE

        From the RetireUp context, extract and list only the **relevant planning
flags** that require advisor attention or action.
        This is a **timeline-based planning system** — your goal is to surface key
visual or structural data such as:
```

```
      - Gaps in income vs expenses
      - Retirement age targeting
      - Rule of 85 tracking
      - Age-based milestones (SS, Medicare, RMDs, QCDs)
      - Previously modeled events (Roth conversions, drawdowns, lump sums)

      When available, always extract specific **years, dollar amounts, frequency, or
account sources** as presented.
      **Do not use "Yes/No" alone** — pull exact numbers if shown.

      ---

      ### 🚨 Advisor Flag Categories (Extract Detailed Data if Present)

      Each flag must include:

      - **Flag Title**
      - **Trigger Explanation (based strictly on shown values)**
      - **Advisor Action Recommendation**
      - **Detailed Values (if visible): year, amount, account, age, etc.**

      ---

      #### 1. 💵 Timeline-Based Cash Flow Gap
      - Trigger only if: A future year explicitly shows income < expenses or a
negative cash flow projection.
      > Reasoning: Pull the year(s) of shortfall and any labeled income sources/gaps.
Example: "2035 shows -$12,000 gap after annuity income."

      ---

      #### 2. 🎯 Retirement Age Target Noted
      - Trigger only if: A specific retirement age or year is shown or labeled in the
timeline.
      > Reasoning: Extract the stated age/year, income source transitions, or
employer plan stops, if included.

      ---

      #### 3. 🧮 Rule of 85 Referenced (if shown)
      - Trigger only if: Timeline flags eligibility under Rule of 85, especially for
public sector employees.
      > Reasoning: Quote exact age/tenure and triggering plan year if labeled. Do not
assume eligibility.

      ---

      #### 4. 🔁 Roth Conversion Scenario Modeled
      - Trigger only if: Timeline shows conversion events or amounts from
tax-deferred to Roth.
      > Reasoning: Extract conversion years, account sources, and dollar amounts
(e.g., "2026 Roth Conversion: $35,000 from Trad. IRA").
```

```
      > Advisor Action: `"Modeled previously - Advisor check-in recommended."`

      ---

      #### 5. 📉 High-Tax Year Identified
      - Trigger only if: Timeline shows spike in tax liability or taxable income for a specific year.
      > Reasoning: Capture exact year and cause if labeled (e.g., large RMD, sale event, benefit overlap).

      ---

      #### 6. 👴 Age-Based Milestone Visuals (Only if explicitly shown)

      % if input['client_applicability']['applicable_for_social_security_benefits']:
      - **Social Security Start Modeled**
      > Reasoning: Include projected monthly benefit, age or year started, and if spousal/own benefit.
      % endif

      % if input['client_applicability']['applicable_for_medicare_benefits']:
      - **Medicare Enrollment Noted**
      > Reasoning: Only if visualized with age, premium modeling, or HSA ending. Quote data if present.
      % endif

      % if input['client_applicability']['applicable_for_rmds']:
      - **RMDs Visualized**
      > Reasoning: Pull first RMD year, withdrawal amount(s), or source account(s) if plotted in timeline.
      % endif

      % if input['client_applicability']['applicable_for_qcds']:
      - **QCD Mentioned or Shown**
      > Reasoning: If shown in gifting stream, extract IRA source, amount/year, and recipient if labeled.
      % endif

      ---

      #### 7. 📝 Previously Modeled Events
      - Trigger only if: Roth conversions, lump sums, annuity withdrawals, or required drawdowns are clearly plotted.
      > Advisor Action: `"Modeled previously - Advisor check-in recommended for implementation follow-up."`
      > Details: Extract year, amount, and event type as presented.

      ---

      ⚠️ For any category where no visual or written RetireUp data exists:
      → `"No action needed based on current RetireUp data."`
```

```
        ---

        ### 📥 Input Variables

        - `client_profile`:
        {input['client_profile']}

        - `client_applicability`:
        {input['client_applicability']}

        - `retire_up_context`:
        {input['retire_up_context']}
        """

        result = model.invoke(prompt).content

        logger.info(f"RetireUp Planning Analysis: {result}")

        return {'retireup_planning_analysis': result}
    except Exception as e:
        logger.error(f"Error in retire_up_planning_analysis: {e}")
        raise

def planning_and_retirement_trigger_graph(
    right_capital_context: str,
    retire_up_context: str
):
    """
    This function is used to generate the planning and retirement trigger graph.
    """

    graph_builder = StateGraph(PlanningAndRetirementTriggerGraph)

    graph_builder.add_node("rightcapital_planning_data_node",
rightcapital_planning_data)
    graph_builder.add_node("rightcapital_planning_analysis_node",
rightcapital_planning_analysis)
    graph_builder.add_node("retire_up_planning_analysis_node",
retire_up_planning_analysis)

    if right_capital_context:
        graph_builder.add_edge(START, "rightcapital_planning_data_node")
        graph_builder.add_edge("rightcapital_planning_data_node",
"rightcapital_planning_analysis_node")
        graph_builder.add_edge("rightcapital_planning_analysis_node", END)

    if retire_up_context:
        graph_builder.add_edge(START, "retire_up_planning_analysis_node")
        graph_builder.add_edge("retire_up_planning_analysis_node", END)

    if not right_capital_context and not retire_up_context:
        graph_builder.add_edge(START, END)
```

```
    return graph_builder.compile()
#household_service = HouseholdLookupService()
#client_name = household_service.find_household_by_individual(input['client_name'])
```

```
from datetime import datetime
import os
from typing import Any, TypedDict

from langchain_openai import AzureChatOpenAI
import requests

from api.app.config.logger import logger
from langgraph.graph import START, END, StateGraph

from api.app.helpers.web_searches_helper import get_percent_change,
get_ytd_percent_change, serper_url_retriever_tool_func

class EstateInsuranceAndGovernmentGraph(TypedDict):
    client_name: str

    client_profile: Any
    client_applicability: Any
    redtail_context: Any

    estate_insurance_and_miscellaneous_info: str
    government_source_logic_2025: str
    market_update: Any

def get_estate_insurance_and_miscellaneous_info(
    input: EstateInsuranceAndGovernmentGraph
):
    """
    This function is used to get the estate insurance and miscellaneous info.
    """
    try:
        model = AzureChatOpenAI(
            azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
            api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
            temperature=0,
            max_tokens=None,
            timeout=None,
```

```
            max_retries=2
        )

        today = datetime.now().strftime("%Y-%m-%d")

        prompt = f"""
        You are a financial planning AI assistant.

        Your task is to **analyze the full historical Redtail context**, specifically extracting relevant data from all prior **client meetings (reviews)**. You must **compile detailed structured data**, organized **by meeting date**, for any item related to:

        - Estate documents
        - Beneficiaries
        - Long-Term Care (LTC) or Medicare planning
        - Trust distributions or activity
        - Life insurance status or changes

        You must also cross-check the findings with the **client's current profile** and **eligibility**:
        - Is the client eligible for tax-free withdrawals?
        - Are they near or over the age of Medicare or RMD?
        - Do they own trusts, IRAs, or other vehicles requiring regular planning?

        ⚠️ Do not assume or invent information.
        ⚠️ Use only explicit notes found in meetings, activities, or advisor entries.

        ---

        📅 Today's Date: **{today}**

        ---

        ### 🔎 Extraction Instructions

        From every **client review** or meeting in the Redtail context:

        #### 1. 🏛️ Estate Documents
        - Extract any mention of:
        - Wills, Trusts, POAs, Healthcare directives
        - Whether they were reviewed, updated, or flagged as incomplete
```

```
- Include:
- **Date of Meeting**
- **Status/Updates Mentioned**
- **Advisor Comments or Tasks**
- If missing → `"No estate documentation discussion"`

#### 2. 👥 Beneficiary Information
- Look for:
- Any mention of confirmation, change, or review
- Life events (e.g., birth, death, marriage) triggering a re-confirmation
- Include:
- **Date of Mention**
- **Which accounts were referenced**
- **Beneficiary names, if noted**
- If missing → `"No beneficiary review found"`

#### 3. 🩺 LTC / Medicare Planning
% if input['client_applicability']['applicable_for_medicare_benefits']:
- Extract:
- Any Medicare enrollment, review, renewal, premium changes
- LTC policy discussions or pricing updates
- Include:
- **Date of Discussion**
- **Carrier, premium, or strategy if stated**
- If missing → `"No Medicare or LTC planning discussion"`
% endif

#### 4. 🧱 Trust Distributions or Use
- Look for:
- Mention of trusts in use (gifting, income, protection)
- Any distribution that occurred or was planned
- Include:
- **Date of Meeting**
- **What was stated about the trust**
- **Follow-up tasks or unresolved issues**
- If missing → `"No trust-related activity found"`

#### 5. 🛡️ Life Insurance Status
- Check for:
- Term policy expirations
- Premium reviews or policy changes
- Recommendations to re-quote or replace
```

```
        - Include:
        - **Date of Mention**
        - **Policy type, carrier, amount (if stated)**
        - If missing → `"No life insurance review recorded"`

        ---

        ### ✅ Output Format

        For each category, organize the data like this:

        ```markdown
        ### [Category Name]

        #### 🗓️ [Date: YYYY-MM-DD]
        - [Summary of finding or note]
        - [Advisor or client follow-up task, if any]

        If no information is found for a category:
        → `"No data found. Advisor review recommended."`

        ---

        ### 📥 Input Variables

        * `client_profile`:
        {input['client_profile']}

        * `client_applicability`:
        {input['client_applicability']}

        * `redtail_context`:
        {input['redtail_context']}
        """

        response = model.invoke(prompt).content

        logger.info(f"Estate Insurance and Miscellaneous Info: {response}")

        return {'estate_insurance_and_miscellaneous_info': response}
    except Exception as e:
        logger.error(f"Error in get_estate_insurance_and_miscellaneous_info: {e}")
```
```

```
        raise

GOVERNMENT_SOURCE_LOGIC_2025 = """
Here is a **detailed 2025 regulatory reference summary** with **rules, limits, and
eligibility requirements** from **IRS.gov**, **SSA.gov**, and **Medicare.gov**,
organized by area. Every section includes:

* 💡 **Key thresholds/rules for 2025**
* 📌 **Eligibility or application requirements**
* ⚠️ "**Check required**" flags where data like DOB, income, or account info is
necessary

---

## 🏛️ IRS – 2025 Rules, Limits, and Thresholds

### 🧾 1. **Federal Income Tax Brackets (2025)**

* Based on filing status: Single, Married Filing Jointly (MFJ), Head of Household,
etc.
* **Bracket ranges adjusted for inflation**

**Single (Taxable Income)**
- 10%: $0 – $11,925
- 12%: $11,926 – $48,475
- 22%: $48,476 – $103,350
- 24%: $103,351 – $197,300
- 32%: $197,301 – $250,525
- 35%: $250,526 – $626,350
- 37%: > $626,350

**Married Filing Jointly (Taxable Income)**
- 10%: $0 – $23,850
- 12%: $23,851 – $96,950
- 22%: $96,951 – $206,700
- 24%: $206,701 – $394,600
- 32%: $394,601 – $501,050
- 35%: $501,051 – $751,600
- 37%: > $751,600

📌 **Requirements**:
* Report all taxable income.
```

Married Filing Jointly:
▪ Base Deduction → $30,000
▪ Age 65+ or Blind → + $1,600 per spouse
Individual (Single):
▪ Base Deduction → $15,000
▪ Age 65+ or Blind → + $2,000
Head of Household:
▪ Base Deduction → $22,500
▪ Age 65+ or Blind → + $2,000
◦ Action:
▪ Use total deduction amount to estimate taxable income before executing tax recommendations.
▪ Higher deduction limits in 2025 provide added flexibility for managing taxable income.

⚠️ **Check required** if: filing status is unclear, or age not provided.

🎁 2. **Qualified Charitable Distributions (QCDs)**

* **Limit (2025):** **$108,000** per individual from IRAs (indexed).
* **Counts toward RMD** if age ≥ 70½.

📌 **Requirements**:
* Must be **70½ or older** on the date of distribution.
* Trustee-to-charity **direct transfer** to a qualified §501(c)(3).
* Applies to **traditional IRAs** (not 401(k)s, donor-advised funds, or most private foundations).

⚠️ **Check required** if: account isn't an IRA, DOB missing, or charity eligibility uncertain.

📉 3. **Required Minimum Distributions (RMDs)**

* **Start age:** **73** (born **1951–1959**); **75** (born **1960 or later**).
* RMD = Prior Dec 31 balance ÷ life expectancy divisor (IRS Uniform Lifetime Table).

📌 **Requirements**:
* Applies to **traditional IRAs**, 401(k), 403(b), etc.

```
* First RMD due by **April 1** of the year after reaching the applicable RMD age; subsequent by **Dec 31** annually.
* Use **Pub. 590-B / IRS tables** for divisors.

⚠️ **Check required** if: DOB, account ownership/beneficiary or plan type missing.

---

## 🧓 SSA – 2025 Social Security Rules

### 📆 4. **Full Retirement Age (FRA)**

| Birth Year | FRA            |
| ---------- | -------------- |
| 1955       | 66 & 2 months  |
| 1959       | 66 & 10 months |
| 1960+      | **67**         |

📌 **Requirements**:
* File with SSA (online/in person).
* Can claim as early as **62** (reduction applies).
* Delaying past FRA increases benefit up to **age 70**.

⚠️ **Check required** if: DOB missing/unclear.

---

### 💵 5. **Earnings Limits Before FRA (2025)**

* **Under FRA all year:** **$23,400** annual limit; $1 withheld for each $2 over.
* **Year reaching FRA:** **$62,160** limit (applies only to months before FRA); $1 withheld for each $3 over.
* **At/after FRA:** No earnings limit.

📌 **Requirements**:
* Applies to workers under FRA receiving benefits.

⚠️ **Check required** if: income type/amount unknown or not wage/SE income.

---

### 📈 6. **COLA (Cost-of-Living Adjustment)**
```

* **2025 COLA:** **+2.5%** to monthly benefits starting Jan 2025.

🔢 7. **Credits Needed for Retirement Benefits**

* **2025 earnings per credit:** **$1,810**
* **Max per year:** **4 credits = $7,240** in earnings

📌 **Requirements**:
* **40 credits** (~10 years of work) generally required for retirement benefits.

🏥 Medicare – 2025 Thresholds and Premiums

🏨 8. **Medicare Part A (Hospital)**

Category	Monthly Premium (2025)
≥40 quarters (premium-free)	$0
30–39 quarters	**$285**
<30 quarters (voluntary Part A)	**$518**

📌 **Requirements**:
* Based on your (or spouse's) covered work history.

🩺 9. **Medicare Part B (Medical)**

* **Standard monthly premium (2025):** **$185.00**
* **Annual deductible (2025):** **$257**

📌 **Income-Related Monthly Adjustment Amount (IRMAA) – 2025 (uses 2023 MAGI, 2-year lookback):**

2023 MAGI (Individual)	2023 MAGI (Married Filing Jointly)	Added Monthly Premium (IRMAA)

```
| ≤ $106,000            | ≤ $212,000                          | $0.00
|
| $106,001 – $133,000   | $212,001 – $266,000                 | $74.00 for Part B;
$13.70 for Part D |
| $133,001 – $167,000   | $266,001 – $334,000                 | $185.00 for Part B;
$35.30 for Part D |
| $167,001 – $200,000   | $334,001 – $400,000                 | $295.90 for Part B;
$57.00 for Part D |
| $200,001 – $499,999   | $400,001 – $749,999                 | $406.80 for Part B;
$78.60 for Part D |
| ≥ $500,000            | ≥ $750,000                          | $432.50 for Part B;
$85.80 for Part D |

📌 **Requirements**:
* MAGI = AGI + tax-exempt interest.
* SSA uses a **2-year lookback** (2023 return → 2025 premiums).
* IRMAA applies separately for **Part B** and **Part D**; surcharges auto-deducted
from Social Security or billed by CMS.

⚠️ **Check required** if: MAGI or filing status missing, or if an appeal
(life-changing event) may adjust income basis.
---

### 💊 10. **Medicare Part D (Prescription)**

* **National base premium (2025):** **$36.78** (used for penalty/IRMAA calculations)
* **IRMAA surcharge (2025):** +$13.70 / +$35.30 / +$57.00 / +$78.60 / +$85.80, by the
same MAGI tiers as Part B.
* **Out-of-Pocket (OOP) cap (2025):** **$2,000** annual maximum under IRA Part D
redesign.

---

### 🤝 11. **Medicare Savings Programs (MSPs) – 2025 Federal Limits**
*(States may use higher limits or different counting rules.)*

| Program | **Monthly Income (Single)** | **Monthly Income (Married)** | **Resource
Limit (Single)** | **Resource Limit (Married)** | Covers |
| --- | ---:| ---:| ---:| ---:| --- |
| **QMB** | $1,325 | $1,783 | $9,660 | $14,470 | Part A & B premiums **and** most
cost-sharing |
| **SLMB** | $1,585 | $2,135 | $9,660 | $14,470 | Part B premium |
```

```
| **QI** | $1,781 | $2,400 | $9,660 | $14,470 | Part B premium (first-come) |
| **QDWI** | $5,302 | $7,135 | $4,000 | $6,000 | Part A premium (if disabled/working
and lost premium-free Part A) |

📌 **Requirements**:
* Apply via your **state Medicaid** agency; names/process vary by state.

---

## ✅ Summary Table (2025)

| Area                       | Rule/Item           | 2025 Value           |
| -------------------------- | ------------------- | -------------------- |
| Standard Deduction         | Single / MFJ / HOH  | $15,000 / $30,000 / $22,500 |
| Add'l Std Deduction 65+    | Single/HOH; MFJ/MFS | $2,000; $1,600 (per spouse) |
| QCD Limit                  | Per IRA owner       | **$108,000** |
| RMD Start Age              | Born 1951–59 / 1960+| **73** / **75** |
| Social Security COLA       | Annual increase     | **+2.5%** |
| Earnings Limit (Pre-FRA)   | Annual              | **$23,400** (FRA year: **$62,160**)
|
| SS Credit Amount           | Per credit / Max    | **$1,810** / **4 credits ($7,240)**
|
| Medicare Part B Premium    | Base monthly        | **$185.00** |
| Part B Deductible          | Annual              | **$257** |
| Part D OOP Cap             | Annual              | **$2,000** |
| Part A Premiums            | 40+ qtrs / 30–39 / <30 | $0 / **$285** / **$518** |

---
"""


def format_government_source_logic_2025(
    input: EstateInsuranceAndGovernmentGraph
):
    """
    This function is used to format the government source logic 2025.
    """

    try:
        # prompt = f"""
        # Please, format the following government source logic 2025 into a markdown
format that is concise, rich in number and details and useful for a Financial Advisor.
```

```
        # {GOVERNMENT_SOURCE_LOGIC_2025}
        # """

        # model = AzureChatOpenAI(
        #     azure_deployment=os.getenv("AZURE_OPENAI_LLM_MODEL_V2"),
        #     api_version=os.getenv("AZURE_OPENAI_API_VERSION_V2"),
        #     temperature=0,
        #     max_tokens=None,
        #     timeout=None,
        #     max_retries=2
        # )

        # response = model.invoke(prompt).content

        # logger.info(f"Formatted Government Source Logic 2025: {response}")

        return {'government_source_logic_2025': GOVERNMENT_SOURCE_LOGIC_2025}
    except Exception as e:
        logger.error(f"Error in format_government_source_logic_2025: {e}")
        raise

def extract_market_update(
    input: EstateInsuranceAndGovernmentGraph
):
    """
    This function is used to extract the market update.
    """

    # web_search_result = serper_url_retriever_tool_func(
    #     query=f"""
    #     Search the Market Update from today {datetime.now().strftime("%Y-%m-%d")}
    #     1. US Total Market
    #     2. International Total Market
    #     3. Total Bond Market
    #     """
    # )

    try:

        def get_token():
            url = f"{os.getenv('VISION_AGENTS_API')}/api/v1/auth/token"
            headers = {
```

```
                "accept": "application/json",
                "Content-Type": "application/json"
            }
            data = {"security_key": os.getenv('SECURITY_KEY')}
            response = requests.post(url, headers=headers, json=data)
            response.raise_for_status()
            token = response.json()["access_token"]
            print("Generated Token:", token)
            return token

        token = get_token()

        headers = {
            "accept": "application/json",
            "authorization-key": f"Bearer {token}",
            "Content-Type": "application/json"
        }

        logger.info(f"Fetching YTD stock values from API")

        from requests.adapters import HTTPAdapter
        from urllib3.util.retry import Retry

        # Configure retry strategy
        retry_strategy = Retry(
            total=3,  # number of retries
            backoff_factor=5,  # will wait 5s, 10s, 20s between retries
            status_forcelist=[500, 502, 503, 504],  # HTTP status codes to retry on
        )

        # Create session with retry strategy
        session = requests.Session()
        adapter = HTTPAdapter(max_retries=retry_strategy)
        session.mount("http://", adapter)
        session.mount("https://", adapter)

        try:
            response = session.get(

f"{os.getenv('VISION_AGENTS_API')}/api/v1/data-sources/ytd-stock-values",
                headers=headers
            )
```

```
            response.raise_for_status()
            ytd_values = response.json()
        except requests.exceptions.RequestException as e:
            logger.error(f"Failed to fetch YTD stock values after retries: {str(e)}")
            raise

        raw_changes = {
            "US Total Market (ITOT)": ytd_values.get("itot", "N/A"),
            "International Total Market (IXUS)": ytd_values.get("ixus", "N/A"),
            "Total Bond Market (AGG)": ytd_values.get("agg", "N/A"),
            "US Small Cap (IJR)": ytd_values.get("ijr", "N/A"),
            "US Mid Cap (IJH)": ytd_values.get("ijh", "N/A"),
            "US Ten Year Yield (US10Y)": ytd_values.get("tnx", "N/A"),
            "Federal Funds Rate": ytd_values.get("fed_funds_rate", "N/A"),
        }

        logger.info(f"YTD stock values: {raw_changes}")

        return {'market_update': raw_changes}
    except Exception as e:
        logger.error(f"Error in extract_market_update: {e}")
        raise

def merge_node(state: EstateInsuranceAndGovernmentGraph):
    required_keys = [
        "estate_insurance_and_miscellaneous_info",
        "government_source_logic_2025",
        "market_update"
    ]

    missing_keys = [k for k in required_keys if k not in state]
    if missing_keys:
        raise ValueError(f"Waiting for keys: {missing_keys}")
    return state

def estate_insurance_and_government_graph(
):
    """
    This function is used to generate the estate insurance and government graph.
    """

    graph_builder = StateGraph(EstateInsuranceAndGovernmentGraph)
```

```
    graph_builder.add_node("merge_node", merge_node, defer=True)

    graph_builder.add_node("estate_insurance_and_miscellaneous_info_node",
get_estate_insurance_and_miscellaneous_info)
    graph_builder.add_node("government_source_logic_2025_node",
format_government_source_logic_2025)
    graph_builder.add_node("extract_market_update_node", extract_market_update)

    graph_builder.add_edge(START, "estate_insurance_and_miscellaneous_info_node")
    graph_builder.add_edge(START, "government_source_logic_2025_node")
    graph_builder.add_edge(START, "extract_market_update_node")

    graph_builder.add_edge("estate_insurance_and_miscellaneous_info_node",
"merge_node")
    graph_builder.add_edge("government_source_logic_2025_node", "merge_node")
    graph_builder.add_edge("extract_market_update_node", "merge_node")

    graph_builder.add_edge("merge_node", END)

    return graph_builder.compile()
```

```
CASH_FLOW = """
# **Cash Flow**

## **Cash Flow**

- **Review Income / Current Income**
 - Evaluate income sources (salary, pension, distributions, Social Security, etc.).
 - Include contributions to ALL accounts if available - all contributions are part of
income flow.
 - **IMPORTANT**: Consider both account balances AND contributions YTD (Year-to-Date),
not just balances.
 - **IMPORTANT**: Include Rollovers YTD as part of income flow.
 - Determine if there is a **need for additional income**.
 - Consider projections for current year and next.
 - Key questions:
   - Will employment/consulting continue or end?
   - Timing of Social Security decision?
   - Is current monthly distribution sufficient?

---
```

- **Map Out Expenses**
 - Project **expenses for remainder of current year and into the following year**.
 - Include both recurring and non-recurring items:
   - Household (utilities, renovations, maintenance).
   - Travel/vacations.
   - Vehicle-related (purchase, replacement, repairs).
   - Lifestyle updates (furniture, hobbies).
   - Gifting/charitable donations.
 - Key questions:
   - Any upcoming large one-time expenses?
   - How should they be funded (cash vs. investment withdrawals)?
   - Any ongoing new commitments?

- **Cash on Hand**
 - Review Cash Allocation (from Black Diamond, if available) - show as percentage (%).
 - Ensure reserves are sufficient for planned needs.
 - Consider yield from short-term holdings.
 - Key questions:
   - Is liquidity adequate for upcoming expenses?
   - Should allocation between cash and investments be adjusted?

- **Review Outcomes/Deductions**
 - Evaluate all outflows and deductions affecting net cash flow.
 - Include payroll deductions and withholdings:
   - SEP contributions (ONLY SEP contributions are deductions, if applicable).
   - Federal and State tax withholdings.
   - Insurance premiums (health, life, disability).
   - Other automatic deductions.
 - Key questions:
   - Are deductions optimized for tax efficiency?
   - Should SEP contribution rates be adjusted (if applicable)?
   - Are withholdings adequate to avoid underpayment penalties?
 - **IMPORTANT**: Other account contributions (401(k), IRA, etc.) are part of Income, NOT Outcomes/Deductions. Distributions/withdrawals are part of Income, NOT Outcomes/Deductions.

```
## 🔹 **Domain-Specific Rules/Patterns for Cash Flow**

- **Always four anchors**: Income → Expenses → Cash on Hand → Review
Outcomes/Deductions.
- **Timeline reference**: "remainder of [current year] and into [next year]".
- **Questions included**: prompts for advisor-client discussion.
- **Use hierarchical bullets consistently** with proper Markdown indentation for clear
structure.

- **CRITICAL - Outcomes/Deductions**:
 - ONLY SEP contributions are deductions (if applicable).
 - DO NOT include other account contributions - those are part of Income.
 - DO NOT include distributions/withdrawals - those are part of Income.
 - DO NOT include Rollovers - those are part of Income.

---

## 🔹 **Sample Format Examples for Cash Flow - Placeholders must be replaced with the
actual data**

### **1. Cash Flow**

- **Cash Flow**
 - Review Income → Any need for additional income?
   - Projected to make around [Insert Projected Income] for [Year]
 - Map out expenses for remainder of [Year] and into [Next Year]
   - Potential items: [House Renovations], [Trips planned], [Large Purchases]
 - Cash on Hand → Cash Allocation: [X%] (from Black Diamond)

---

- **Cash Flow**
 - Review Current Income → Any need for additional income on top of
distributions/pension?
   - Currently drawing [Insert Monthly Distribution] each month
     - Will [Client/Spouse] be done working for good?
 - Map out expenses for remainder of [Year] and into [Next Year]
   - [Travel Plans], [Home Updates], [Vehicle Replacement]
 - Cash on Hand → Cash Allocation: [X%] yielding ~[%]

---
```

```
- **Cash Flow**
 - Review Current Income → Any need for additional income?
   - Currently gifting [X Amount] per month to [Recipient]
     - Will total ~$[Annual Total] by end of the year
 - Map out expenses for remainder of [Year] and into [Next Year]
   - [Furniture Updates], [Other Planned Expenses]
 - Cash on Hand → Cash Allocation: [X%] yielding ~[%]

---

- **Cash Flow**
 - Review Current Income → Update
   - [Salary/Distribution Source 1: $XXX,XXX]
   - [Salary/Distribution Source 2: $XX,XXX]
 - Expenses → Any need for additional income?
   - Map out expenses for remainder of [Year] and heading into [Next Year]
 - Cash on Hand → Cash Allocation: [X%]

---
"""

INVESTMENT_PLANNING = """
# **Investment Planning**

## **Investment Planning / Investments / Investment Portfolio**

- **Liquidity & Cash Coming Due**
 - Identify cash readily available or maturing soon.
 - Note amounts in money market or CDs with maturity dates.
 - Consider reinvestment strategies when funds free up.

---

- **Investment Approach & Portfolio Design**
 - Define the stance from **Black Diamond profile ONLY** (Conservative, Moderate,
Moderate Aggressive, Aggressive).
 - Portfolio designed to:
   - Achieve **True Diversification** (across asset classes and strategies).
   - Provide **Smoother Returns** (non-correlated strategies).
   - Capture **Opportunities** (growth potential, rebalancing).
   - Ensure **Risk Mitigation** (protect capital, not just beat benchmark).
```

```
---

- **Forward-Looking Actions**
 - Continue laddering strategy until cash is needed.
 - Monitor Fed moves (potential rate cuts, bull market implications).
 - Plan around specific events (vesting schedules, RSUs, retirement timeline).

---

## 🔹 **Domain-Specific Rules/Patterns for Investment Planning**

- **CRITICAL**: Investment Portfolio data MUST ALWAYS come from **Black Diamond
ONLY**. NEVER mention or reference Nitrogen or RightCapital for investment portfolio
information.
- Always start with **Liquidity/Cash** → what funds are available or coming due?
- Always include **Portfolio stance from Black Diamond profile ONLY** (Conservative,
Moderate, Moderate Aggressive, Aggressive) - NEVER use profiles from Nitrogen or
RightCapital.
- Use the **portfolio design** block with the 4 constants:
 - Diversification
 - Smoother Returns
 - Opportunities
 - Risk Mitigation
- Include Risk Profile (From Black Diamond)
- Close with **Forward-Looking Actions** (laddering, RSUs, expected changes).
- Use consistent hierarchical indentation for clear structure.

---

## 🔹 **Sample Format for Investment Planning - Placeholders must be replaced with the
actual data**

### **2. Investment Planning**

- **Investment Planning**
 - Currently have cash readily available and coming due in the next months
   - ~$[Amount] maturing on [Date]
   - ~$[Amount] currently in money market earning ~[%]
 - With potential rate/environment changes → Consider reinvesting into the market
   - Be strategic → Invest in positions with downside protection ("built-in floor")
     - Potential to capture equity upside while mitigating volatility
```

```
 - Possible tools: [Buffers] | [Hedged Equity] | [Treasuries Ladders] | [Structured
Notes]

---

- **Investment Planning**
 - Maintaining a [Conservative / Moderate / Moderate Aggressive / Aggressive] approach
(**from Black Diamond profile ONLY**)
   - Market has [recent trend] → Potential to harvest gains / prepare for volatility
 - Investment Approach
   - True Diversification – beyond stocks & bonds, across asset classes
   - Smoother Returns – non-correlated strategies to reduce volatility
   - Opportunities – multiple return streams, growth, and rebalancing
   - Risk Mitigation – focus on protecting capital, not just benchmarks

---

- **Investment Portfolio**
 - Currently positioned at a [Conservative / Moderate / Moderate Aggressive /
Aggressive] profile (**from Black Diamond ONLY**)
   - Portfolio design:
     - Diversification
     - Smoother Returns
     - Opportunities
     - Risk Mitigation
 - Market outlook: [bull market / Fed cuts / volatility expectations]
   - Example: "If Fed cuts, potential for market pop"

---

- **Investment Planning**
 - Review future actions
   - RSUs or stock grants vesting in [Year] → plan diversification strategy
   - Adjust laddering or allocations as needs evolve

---
"""

TAX_PLANNING = """
# **Tax Planning**

## **Tax Planning**
```

```
- **Tax Projections**
 - **CRITICAL**: Always evaluate client's filing status (Single vs Married Filing Jointly) as tax brackets, capital gains rates, and Social Security taxation thresholds differ significantly.
 - Provide income and tax estimates for current and next year.
 - Break down by income sources: [Wages], [Interest], [Dividends], [Pensions], [Social Security], [Capital Gains/Losses], [QCDs], etc.
 - **IMPORTANT**: Only include Taxable Income section if there are actual distribution data. Otherwise, omit this section entirely.
 - If distributions exist, classify them automatically by type:
   - **Taxable Distributions**:
     - Traditional IRA distributions
     - 401(k), 403(b), 457(b) distributions
     - Non-qualified account distributions
     - Realized capital gains
     - SEP IRA distributions
   - **Non-Taxable Distributions**:
     - Roth IRA qualified distributions
     - HSA distributions used for qualified medical expenses
     - TODI principal distributions
     - Return of basis
 - Show **Taxable Income** and **Estimated Federal/State Tax Due or Refund**.
 - Compare projected liability vs. prior year.

---

- **Medicare & Surcharges (IRMAA)**
 - Filing Status: [Single / Married Filing Jointly]
 - Current MAGI: $[Current_MAGI]
 - Surcharge thresholds (for [Filing_Status]):
   - Threshold 1: $[Threshold_1] → +$[Additional_Cost_1]/month
   - Threshold 2: $[Threshold_2] → +$[Additional_Cost_2]/month
 - Action: adjust distributions/conversions to avoid crossing into higher surcharge brackets.

---

- **HSA / FSA Contributions**
 - Annual limit: $[HSA_FSA_Limit].
 - YTD contributions: $[YTD_Contribution].
 - Remaining room: $[Remaining_Contribution].
```

```
 - Advisor note: stop contributions if limits reached.

---

- **Client To-Do's & Recommendations**
 - Provide [Tax_Returns / Paystubs / Premium_Payment_Info].
 - Recommendation: adjust withholdings by $[Adjustment_Amount].
 - Questions to clarify:
   - How much are you receiving from pensions?
   - How much are you distributing from IRAs?
   - Are you withholding Social Security?
   - Do you expect additional [Bonus/Commission]?

---

## 🔹 **Domain-Specific Rules/Patterns for Tax Planning**

- Always include **Tax Projections** with breakdown by income sources. **CRITICAL**: Always evaluate client's filing status (Single vs Married Filing Jointly) as tax brackets, capital gains rates, and Social Security taxation thresholds differ significantly between filing statuses.
- **CRITICAL**: Only include distribution details if actual distribution data exists. If no distributions, omit the distribution section entirely.
- When distributions exist, always classify them into:
 - **Taxable Distributions**: Traditional IRA, 401(k)/403(b)/457(b), non-qualified accounts, realized capital gains, SEP IRA
 - **Non-Taxable Distributions**: Roth IRA (qualified), HSA (medical), TODI principal, return of basis
- **CRITICAL**: Taxable Income is NOT the total of all withdrawals. Exclude non-taxable distributions from the Taxable Income calculation. Only Taxable Distributions should be included in Taxable Income.
- Include **Medicare & Surcharges** (MAGI thresholds and surcharge brackets) when applicable. **CRITICAL**: Always evaluate client's filing status (Single vs Married Filing Jointly) as IRMAA thresholds differ significantly between filing statuses.
- Include **HSA/FSA Contributions** when applicable.
- **Do NOT include RMDs here** - RMDs belong in a separate section or in Retirement Planning.
- Use consistent hierarchical indentation for clear structure.

---
```

```
##  🔹 **Sample Format for Tax Planning - Placeholders must be replaced with the actual data**

### **3. Tax Planning**

- **Tax Planning**
 - [Year] Tax Projection
   - Total Income: $[Total_Income]
   - Wages: $[Wages]
   - Interest: $[Interest]
   - Dividends: $[Dividends]
   - Pension: $[Pension]
   - Taxable Social Security: $[Taxable_SS]
   - **Taxable Distributions**:
     - Traditional IRA: $[Trad_IRA_Dist]
     - 401(k)/403(b)/457(b): $[401k_Dist]
     - Realized Capital Gains: $[Cap_Gains]
   - **Non-Taxable Distributions**:
     - Roth IRA (qualified): $[Roth_Dist]
     - HSA (medical expenses): $[HSA_Dist]
     - TODI Principal: $[TODI_Principal]
   - Short-Term Gains/Losses: $[ST_Gains_Losses]
   - Long-Term Gains/Losses: $[LT_Gains_Losses]
   - Taxable Income: $[Taxable_Income]
   - Federal Tax Due/Refund: $[Fed_Tax]
   - State Tax Due/Refund: $[State_Tax]
 - Client To-Do's
   - Provide [Premium_Payments / Tax_Returns / Updated_Paystubs]

---

- **Tax Planning** (Example WITHOUT distributions - omit distribution section)
 - [Year] Tax Projection
   - Total Income: $[Total_Income]
   - Wages: $[Wages]
   - Interest: $[Interest]
   - Dividends: $[Dividends]
   - Pension: $[Pension]
   - Taxable Social Security: $[Taxable_SS]
   - Taxable Income: $[Taxable_Income]
   - Federal Tax Due/Refund: $[Fed_Tax]
   - State Tax Due/Refund: $[State_Tax]
```

```
 - Client To-Do's
   - Provide [Premium_Payments / Tax_Returns / Updated_Paystubs]

---

- **Tax Planning**
 - Medicare Surcharge Thresholds (IRMAA)
   - Filing Status: [Single / Married Filing Jointly]
   - Current MAGI: $[Current_MAGI]
   - Threshold 1: $[Threshold_1] (for [Filing_Status]) → +$[Additional_Cost_1]
   - Threshold 2: $[Threshold_2] (for [Filing_Status]) → +$[Additional_Cost_2]
 - Adjusted Plan
   - MAGI after conversion: $[Adjusted_MAGI]
   - Federal Tax Due/Refund: $[Adjusted_Fed_Tax]
   - State Tax Due/Refund: $[Adjusted_State_Tax]

---

- **Tax Planning**
 - HSA & FSA Contributions
   - Limit: $[Contribution_Limit]
   - YTD Contributions: $[YTD_Contribution]
   - Remaining: $[Remaining_Contribution]
 - Advisor Notes
   - Confirm HSA carryover rules
   - Stop FSA contributions if already maxed

---
"""

ESTATE_PLANNING = """
# **Estate Planning**

## **Estate Planning**

- **Document Updates**
 - Ensure all estate documents (Wills, Trusts, POAs, Healthcare Directives) are
updated after key life events ([Marriage / Birth / Divorce / Relocation]).
 - Summarize key designations:
   - Financial Agent/Trustee: [Name]
   - Healthcare Agent(s): [Name(s)]
   - Guardians: [Name(s)]
```

```
   - Beneficiaries: [Name(s)] with allocation [X% / Y%]. The beneficiaries must be presented for each existent client.
 - Age-based restrictions: assets distributed at ages [Age_1] / [Age_2] / [Age_3].
 - Advisor Question: *What updates would you like to make?*

---

- **Account Titling**
 - Review account ownership and update to include Trust as primary or contingent (When applicable)
 - Non-Qualified (NQ) Accounts:
   - Retitle from individual → Trust as contingent (When applicable).
   - Benefit: avoids probate and ensures smooth transfer of assets.
 - Retirement Accounts (Traditional IRA, Roth IRA, SEP IRA, 401(k), 403(b), 457(b), etc.)
 - Tax Considerations:
   - If distributed via Trust → Trust pays taxes before distribution.
   - If distributed directly to child → taxed at child's rate.

---

- **Funeral Expenses**
 - Discuss pre-paying for funeral-related expenses.
 - Option: designate beneficiaries to receive funds to cover costs for each client
 - Benefit: preserves client liquidity ("keep the money in your pocket now").

---

- **Real Estate & Other Assets**
 - Review real estate holdings ([Property_Name / Project]).
 - Current valuation: $[Amount].
 - Return expectations: [X% IRR] with [Y% preferred return].
 - Consider titling into Trust to ensure smooth succession.

---

## 🔹 **Domain-Specific Rules/Patterns for Estate Planning**

- Always start with **document updates** (Trust, POA, Healthcare, Guardians, Beneficiaries).
- **CRITICAL - Trustee/Agent Designation for Married Couples**: If only one spouse is detected as trustee or agent in the data, the system must automatically include both
```

```
spouses (each designating the other as their first trustee or agent) unless there is
an explicit reason in the documents indicating otherwise.
- Always address **account titling** (NQ vs. Qualified accounts → probate avoidance +
tax considerations).
- Include **age-based beneficiary restrictions** when applicable.
- Always raise **funeral/expense pre-pay discussion**.
- Include **real estate or alternative asset planning** when relevant.
- Use consistent hierarchical indentation for clear structure.

---

## 🔹 **Sample Format for Estate Planning - Placeholders must be replaced with the
actual data**

### **4. Estate Planning**

- **Estate Planning**
 - Once [Life_Event] occurs → update estate documents and schedule signing
   - Financial Agent/Trustee: [Name]
   - Healthcare Agents: [Name_1], then [Name_2]
   - Guardians: [Name_1], [Name_2]
   - Beneficiary Info: [Allocation e.g., 50/50 between Children] for each client
   - Age-based restrictions: distributions at ages [Age_1] / [Age_2] / [Age_3]
 - Advisor Question: *What updates would you like to make?*

---

- **Account Titling**
 - Primary ownership remains with client(s), but Trust should be added as contingent
   - NQ Accounts → Retitle from individual to Trust
   - Retirement Accounts (IRA, 401(k), 403(b), etc.):
     - If minor beneficiary → retitle to Trust
     - Once beneficiary reaches 18 → retitle to individual
 - Tax Consideration
   - If distributed via Trust → taxed at Trust level
   - If distributed directly to child → taxed at child's rate

---

- **Funeral Expenses**
 - Discuss pre-paying for expenses
   - Beneficiaries designated to cover costs if needed for each client
```

```
   - Benefit: preserves cash flow today

---

- **Real Estate**
 - [Property_Name / Project] ~ $[Amount] current valuation
 - Expected return: [X% IRR] with [Y% preferred return]
 - Consider Trust titling for estate transfer efficiency

---
"""

RETIREMENT_PLANNING = """
# **Retirement Planning**

## **Retirement Planning**

- **Projection Review**
 - Run income and portfolio projections at multiple return rates ([Rate_1]%,
[Rate_2]%, [Rate_3]%).
 - Compare outcomes for retirement income sustainability.
 - Key questions:
   - Will [Future_Year] look the same income-wise?
   - Are current contributions on track?

---

- **External Retirement Accounts (ONLY if they exist)**
 - **ONLY include this subsection if there are confirmed External Accounts**: 401(k),
403(b), 457(b), 401(a), SIMPLE 401(k), etc. held outside of advisory
 - **If NO external accounts exist → OMIT this entire subsection**
 - **DO NOT include Internal Accounts (IRAs)** - these are managed within advisory
 - When external accounts exist, include:
   - Current value: ~$[Current_Value] as of [Date]
   - Current allocation: [Equity_%] Equities / [Bond_%] Bonds
   - Contribution review: On track to max out for [Year]?
   - Is brokerage link available for tracking?

---

- **Medicare**
 - Maintain taxable income low enough to preserve ACA subsidy eligibility.
```

```
 - Monitor Roth IRA distributions and Medicare impact.
 - Event: [Client_Name] turning 65 on [Date] → Medicare enrollment.
 - Advisor prompt: *Have you started the enrollment process?*

---

- **Social Security**
 - Status: [Status]
 - Rationale: [Rationale]

---

- **Annuities (if applicable)**
 - [Provider] annuity balance: ~$[Annuity_Value].
 - Current earning rate: [X%].
 - Advisor prompt: *How much income do you withdraw annually from this annuity?*

---

## 🔹 **Domain-Specific Rules/Patterns for Retirement Planning**

- Always begin with **Projection Review** (multiple return rates).
- **External Retirement Accounts - CONDITIONAL SUBSECTION**:
 - **ONLY include if External Accounts explicitly exist**: 401(k), 403(b), 457(b), 401(a), SIMPLE 401(k), etc. (held outside of advisory)
 - **If NO external accounts exist → OMIT the entire subsection** - DO NOT force or create placeholder data
 - **DO NOT include Internal Accounts**: Traditional IRA, Roth IRA, SEP IRA, SIMPLE IRA are NOT External Accounts
- Always integrate **Medicare timeline** (age 65, enrollment process, income planning for subsidies). **CRITICAL**: When discussing Medicare subsidies (ACA) or IRMAA surcharges, always evaluate filing status as income thresholds differ for Single vs Married Filing Jointly.
- Always outline **Social Security strategy** (timing, delaying benefits, simplification of tax year). **CRITICAL**: Always evaluate filing status as Social Security taxation thresholds differ for Single vs Married Filing Jointly.
- Include **Annuity discussion** (annual withdrawal, interest rate) only if annuities exist.
- Don't include RMDs here.
- Use consistent hierarchical indentation for clear structure.

---
```

```
## 🔹 **Sample Format for Retirement Planning - Placeholders must be replaced with the actual data**

### **5. Retirement Planning**

**Example 1: WITH External Accounts**

- **Retirement Planning**
 - Review Projections – Variable Rates of Return [Rate_1]%, [Rate_2]%, [Rate_3]%
 - Project out → Will [Future_Year] look the same income-wise?

- **External Retirement Accounts**
 - [Employer_Name] 401(k)/403(b)/457(b)/401(a)/SIMPLE 401(k) – Value: ~$[Current_Value]
   - Current allocation: [Equity_%] Equities / [Bond_%] Bonds
   - Contribution check: On track to max out for [Year]?
   - Brokerage link available for tracking?
 - Advisor Question: Can we get access to monitor this account?

---

**Example 2: WITHOUT External Accounts**

- **Retirement Planning**
 - Review Projections – Variable Rates of Return [Rate_1]%, [Rate_2]%, [Rate_3]%
 - Project out → Will [Future_Year] look the same income-wise?
 - External Accounts: N/A

---

- **Medicare**
 - Keep taxable income low to maintain subsidy eligibility
   - Example: Roth IRA distribution planning
 - [Client_Name] turns 65 on [Date] → Medicare enrollment begins
   - Advisor Question: Have you started enrollment?

---

- **Social Security**
 - Status: [Status]
 - Rationale: [Rationale]
```

```
---

- **Annuity**
 - [Provider] Annuity balance: ~$[Annuity_Value]
 - Current earnings: [Rate_%]
 - Advisor Question: How much annual income are you drawing from this annuity?

---
"""

DOMAIN_SPECIFIC_SECTIONS = f"""

## Cash Flow.
{CASH_FLOW}
## Investment Planning.
{INVESTMENT_PLANNING}
## Tax Planning.
{TAX_PLANNING}
## Estate Planning.
{ESTATE_PLANNING}
## Retirement Planning.
{RETIREMENT_PLANNING}

DON'T CONBINE THOSE SECTIONS WITH THE NEXT SECTIONS. KEEP THEM SEPARATE AND RESPECT
HOW THE STRUCTURE IS PRESENTED. YOU ARE ALWAYS ALLOWED TO PRESENT THE SAME DATA
SEVERAL TIMES ACCORDING TO THE STRUCTURE.
YOU MUST ALWAYS PRESENT THE FOLLOWING SECTIONS IN THE FINAL RESPONSE, DON'T MIX THEM
WITH THE PREVIOUS SECTIONS:
"""

UNDER_35_AGENDA_TEMPLATE = f"""
### **2025 Overall Planning**

#### Last Meeting – Topics & Client Updates
- You must list all the topics that were discussed in the last meeting here. ALWAYS
MUST BE WRITTEN IN PAST.
- You must also list all the client updates from the last meeting here. These updates
may be sentences or questions, but they must always relate to the client’s profile and
personal activities (distinct from the meeting topics). ALWAYS MUST BE WRITTEN IN
PAST.
```

Topics that we will discuss in this meeting - YOU MUST ALWAYS PUT THIS SECTION:
- You must list all the topics that we will discuss in the meeting here.

* "Any changes since we last met?" ← Always put this
* A placeholder for open-ended or major life event discussions:

Client(s) Age(s): The age(s) of the client(s) must be included here.

{DOMAIN_SPECIFIC_SECTIONS}

Market Update:
The Market Updates should go here (AS A TABLE):
- US Total Market (ITOT)
- International Total Market (IXUS)
- Total Bond Market (AGG)
- US Small Cap (IJR)
- US Mid Cap (IJH)
- US Ten Year Yield (US10Y)
- Federal Funds Rate Range

Black Diamond Portfolio Performance Report - Two Separated Tables (MUST ALWAYS BE TABLES, WITHOUT EXCEPTION):

- Portfolio Allocation Table (from Black Diamond, with percentages).
 STRICT RULES:
 • You MUST ONLY extract this from the Black Diamond context.
 • You MUST ALWAYS show ALL asset classes EXACTLY as they appear in Black Diamond, including smaller categories such as International Bonds, Emerging Markets, Alternatives, or any other.
 • You MUST NEVER omit, summarize, merge, collapse, or rename any asset class. Even if an asset class is very small (e.g., <1%), it MUST still be shown as its own row with its exact percentage.
 • Asset class names MUST be copied VERBATIM from Black Diamond.
 • Percentages MUST be copied VERBATIM from Black Diamond.
 • Even if these values are referenced earlier (e.g., inside Investment Planning), you MUST still generate this table here again.

- Account-Level Returns Table (with percentages). REMEMBER THAT THIS BELONGS TO BLACK DIAMOND, AND NEVER MIX THIS WITH THE ACCOUNT BALANCES
 STRICT RULE: This table is mandatory. You MUST generate it here,
 always separated from the Portfolio Allocation table.

Accounts
* Current cash needs and income sources:

YOU MUST INCLUDE the account balances in addition to the overall data here FOR ALL THE ACCOUNTS (organized by client name, account number, account type and model type per account). DO NOT IGNORE ANY ACCOUNT. INCLUDE ALL THE ACCOUNTS. YOU MUST ALWAYS PRESENT THE OWNER'S NAMES OF EACH ACCOUNT. When you present them in a table, put all of those aligned to the right to easily differentiate larger and shorter account balances, so that visually all decimal points can be in the same position. Include the Total Account Balances row inside the same account balances table, properly formatted as the final row, with the label spanning the appropriate columns and the total amount right-aligned in the balance column.
- The accounts must always be presented as a table.
- This table must always be present.
- The table must always include the following columns:
 - Account Number
 - Account Owner
 - Account Type
 - Model Type
 - Balance
- The Total Account Balances row must always be present in the end of the table.

Transactions - Two Mandatory Tables (MUST ALWAYS BE PRESENT, WITHOUT EXCEPTION):

- Both tables must always include: CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU MUST NOT IGNORE ANY OF THEM
- You must always generate two tables: Transactions (YTD) and Transactions (Prior Year), never forget to include them.
- The two tables must always contain (In each of them, for all the existent accounts):
 - Account Number
 - Contributions
 - Withdrawals
 - Rollovers
 - Conversions
 - Federal Tax Withholdings
 - State Tax Withholdings

* Transactions (YTD) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD PRESENT ALL VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A TABLE.
* Transactions (Prior Year) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD

```
PRESENT ALL VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A
TABLE.

### Net Worth: YOU MUST EXTRACT THESE VALUES ONLY FROM RIGHTCAPITAL. IF NO VALUE EXIST
IN THE RIGHTCAPITAL CONTEXT, YOU MUST OMIT THIS SECTION AND NOT INCLUDE ANY VALUE
HERE. YOU SHOULD PRESENT THIS AS A TABLE
- Assets
- Liabilities
- Net Worth

### Beneficiaries (Table(s)) - These must be tables based upon each existent client:
- The table(s) MUST include EXACTLY these columns (in this order):
 • Name
 • Type (Primary / Contingent)
 • % of Shares
- Beneficiary Type MUST be normalized to either "Primary" or "Contingent" only.
- % of Shares MUST be shown as a percentage with the % symbol (e.g., 50%).
"""

# => Client Profile Sections (Include the sections aligned with the client profile) -
EACH SECTION'S CONTENT MUST BE DEVELOPED AS BULLET POINTS. ALL HERE ARE POTENTIAL
SECTIONS, SO INFO. SHOULD BE INCLUDED AND NOT REPEAT THE SAME NAME OF THE SECTION WITH
NO INFO.  IN THE END OF EACH SECTION, YOU MUST INCLUDE THE DATA SOURCE WHERE YOU GOT
THE INFORMATION FROM.
# - **Key focus:** Building habits, establishing financial security, understanding
time is on your side

# - **Budgeting**
#   - Student loan repayment strategies
#   - Credit score building/repair
#   - Cash Flow Planning - What is coming in vs going out
#   - Emergency fund targets (3-6 months)

# - **Retirement Savings - Establishing a good starting point**
#   - Getting started early
#     - Match from your 401k
#     - Maxing your Roth
#     - Allocating any extra cash in general
#   - Utilizing compounding growth

# - **Career & Income Growth**
#   - Understanding Career Path
```

```
#   - Benefits analysis

# - **Insurance Basics - Income Replacement and the fact you will never be healthier
than you are now**
#   - Term life insurance to supplant debts and take care of dependents if the worst
happens
#   - Understanding how much coverage is enough
#   - Taking advantage of what employer offers

# - **Goals - Time Horizon and Risk Tolerance**
#   - Buying a first home
#   - Saving for marriage/kids
#   - Setting yourself up for the future
#   - Not rushing into anything unless it makes sense for you

# - **Tax Planning**
#   - Maximizing deductions/credits
#   - Pre-Tax vs After-Tax

# - **Investing Basics**
#   - Risk tolerance
#   - Building diversified, long-term portfolios
#   - Not seeking a one-hit wonder

# - **Taxes**
#   - Taxable Income (2024)
#   - Federal
#   - State
#   - Bracket

# - **Top Financial Goals**
#   - Needs
#   - Wants
#   - Wishes

# - **Income Planning**
#   - Husband
#   - Wife
#   - Other Income
#   - Total Income
```

```
# - **Insurance Planning**
#   - Husband
#   - Wife

# - **Employer Review**
#   - Retirement Accounts
#   - % of Contributions
#   - Review Allocation
#   - Benefits
#   - Career Path

# ---

FROM_36_TO_49_AGENDA_TEMPLATE = f"""
### **2025 Overall Planning**

#### Last Meeting - Topics & Client Updates
- You must list all the topics that were discussed in the last meeting here. ALWAYS MUST BE WRITTEN IN PAST.
- You must also list all the client updates from the last meeting here. These updates may be sentences or questions, but they must always relate to the client's profile and personal activities (distinct from the meeting topics). ALWAYS MUST BE WRITTEN IN PAST.

#### Topics that we will discuss in this meeting - YOU MUST ALWAYS PUT THIS SECTION:
- You must list all the topics that we will discuss in the meeting here.

* "Any changes since we last met?" ← Always put this
* A placeholder for open-ended or major life event discussions:

Client(s) Age(s): The age(s) of the client(s) must be included here.

{DOMAIN_SPECIFIC_SECTIONS}

Market Update:
The Market Updates should go here (AS A TABLE):
- US Total Market (ITOT)
- International Total Market (IXUS)
- Total Bond Market (AGG)
- US Small Cap (IJR)
- US Mid Cap (IJH)
- US Ten Year Yield (US10Y)
```

- Federal Funds Rate Range

Black Diamond Portfolio Performance Report - Two Separated Tables (MUST ALWAYS BE TABLES, WITHOUT EXCEPTION):

- Portfolio Allocation Table (from Black Diamond, with percentages).
 STRICT RULES:
 • You MUST ONLY extract this from the Black Diamond context.
 • You MUST ALWAYS show ALL asset classes EXACTLY as they appear in Black Diamond, including smaller categories such as International Bonds, Emerging Markets, Alternatives, or any other.
 • You MUST NEVER omit, summarize, merge, collapse, or rename any asset class. Even if an asset class is very small (e.g., <1%), it MUST still be shown as its own row with its exact percentage.
 • Asset class names MUST be copied VERBATIM from Black Diamond.
 • Percentages MUST be copied VERBATIM from Black Diamond.
 • Even if these values are referenced earlier (e.g., inside Investment Planning), you MUST still generate this table here again.

- Account-Level Returns Table (with percentages). REMEMBER THAT THIS BELONGS TO BLACK DIAMOND, AND NEVER MIX THIS WITH THE ACCOUNT BALANCES
 STRICT RULE: This table is mandatory. You MUST generate it here,
 always separated from the Portfolio Allocation table.

Accounts
* Current cash needs and income sources:

YOU MUST INCLUDE the account balances in addition to the overall data here FOR ALL THE ACCOUNTS (organized by client name, account number, account type and model type per account). DO NOT IGNORE ANY ACCOUNT. INCLUDE ALL THE ACCOUNTS. YOU MUST ALWAYS PRESENT THE OWNER'S NAMES OF EACH ACCOUNT. When you present them in a table, put all of those aligned to the right to easily differentiate larger and shorter account balances, so that visually all decimal points can be in the same position. Include the Total Account Balances row inside the same account balances table, properly formatted as the final row, with the label spanning the appropriate columns and the total amount right-aligned in the balance column.
- The accounts must always be presented as a table.
- This table must always be present.
- The table must always include the following columns:
 - Account Number
 - Account Owner
 - Account Type

```
 - Model Type
 - Balance
- The Total Account Balances row must always be present in the end of the table.

### Transactions - Two Mandatory Tables (MUST ALWAYS BE PRESENT, WITHOUT EXCEPTION):

- Both tables must always include: CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU MUST NOT IGNORE ANY OF THEM
- You must always generate two tables: Transactions (YTD) and Transactions (Prior Year), never forget to include them.
- The two tables must always contain (In each of them, for all the existent accounts):
 - Account Number
 - Contributions
 - Withdrawals
 - Rollovers
 - Conversions
 - Federal Tax Withholdings
 - State Tax Withholdings

* Transactions (YTD) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD PRESENT ALL VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A TABLE.
* Transactions (Prior Year) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD PRESENT ALL VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A TABLE.

### Net Worth: YOU MUST EXTRACT THESE VALUES ONLY FROM RIGHTCAPITAL. IF NO VALUE EXIST IN THE RIGHTCAPITAL CONTEXT, YOU MUST OMIT THIS SECTION AND NOT INCLUDE ANY VALUE HERE. YOU SHOULD PRESENT THIS AS A TABLE
- Assets
- Liabilities
- Net Worth

### Beneficiaries (Table(s)) - These must be tables based upon each existent client:
- The table(s) MUST include EXACTLY these columns (in this order):
 • Name
 • Type (Primary / Contingent)
 • % of Shares
- Beneficiary Type MUST be normalized to either "Primary" or "Contingent" only.
- % of Shares MUST be shown as a percentage with the % symbol (e.g., 50%).
"""
```

```
# => Client Profile Sections (Include the sections aligned with the client profile) - EACH SECTION'S CONTENT MUST BE DEVELOPED AS BULLET POINTS. ALL HERE ARE POTENTIAL SECTIONS, SO INFO. SHOULD BE INCLUDED AND NOT REPEAT THE SAME NAME OF THE SECTION WITH NO INFO. IN THE END OF EACH SECTION, YOU MUST INCLUDE THE DATA SOURCE WHERE YOU GOT THE INFORMATION FROM.
# - **Key focus:** Growing assets, being tax efficient, protecting against risks

# - **Retirement Projections – Start getting a feeling for where you are in life**
#   - Increasing contributions toward Retirement Accounts (IRA, 401(k), 403(b), 457(b), etc.)
#   - Pre-Tax vs After-Tax
#   - Catch-up strategies if behind

# - **College Planning**
#   - 529 plan funding strategies
#   - Balancing retirement vs. education savings – how much is enough

# - **Insurance Review**
#   - As you enter your prime working years – make sure you are properly insured for the people around you
#   - Additional insurance options
#     - Long-term care
#     - Disability

# - **Cash Flow Strategies**
#   - Mortgage payoff strategies
#   - Avoiding living outside of your means
#   - Strategizing which is good vs. bad debt

# - **Tax Efficiency**
#   - Maximizing employer benefits (FSA, HSA)

# - **Estate Planning**
#   - Wills, powers of attorney, guardianship for minors

# - **Investment Allocation Review**
#   - Shifting from aggressive growth to balanced risk tolerance as goals become clearer and change

# - **Business/Side Income**
#   - Retirement planning for self-employed
```

```
#   - Business planning

# - **Review Variables**
#   - **Savings for Retirement and Kids**
#     - Retirement Accounts
#     - 529's / UTMA's

# - **Taxes**
#   - Taxable Income (2024)
#   - Federal
#   - State
#   - Tax Bracket

# - **Income Planning**
#   - Husband
#   - Wife
#   - Total Income

# - **Insurance Planning**
#   - Husband
#   - Wife

# - **Employer Review**
#   - Retirement Accounts
#   - % of Contributions
#   - Review Allocation
#   - Benefits
#   - Career Path

# - **Estate Planning (Documents on file)**
#   - Wills
#   - Healthcare POAs
#   - Durable POAs
#   - Revocable Trust
#   - Beneficiaries (LIST ALL THE BENEFICIARIES)

# - **Retirement Projections**
#   - Projected Retirement Dates
#   - Review Projections
#   - Rate of Return - Conservative
#   - Rate of Return - Aggressive
#   - Timeline & trend
```

```
# ---


FROM_50_TO_61_AGENDA_TEMPLATE = f"""
### **2025 Overall Planning**

#### Last Meeting – Topics & Client Updates
- You must list all the topics that were discussed in the last meeting here. ALWAYS MUST BE WRITTEN IN PAST.
- You must also list all the client updates from the last meeting here. These updates may be sentences or questions, but they must always relate to the client’s profile and personal activities (distinct from the meeting topics). ALWAYS MUST BE WRITTEN IN PAST.

#### Topics that we will discuss in this meeting - YOU MUST ALWAYS PUT THIS SECTION:
- You must list all the topics that we will discuss in the meeting here.

* "Any changes since we last met?" ← Always put this
* A placeholder for open-ended or major life event discussions:

Client(s) Age(s): The age(s) of the client(s) must be included here.

{DOMAIN_SPECIFIC_SECTIONS}

Market Update:
The Market Updates should go here (AS A TABLE):
- US Total Market (ITOT)
- International Total Market (IXUS)
- Total Bond Market (AGG)
- US Small Cap (IJR)
- US Mid Cap (IJH)
- US Ten Year Yield (US10Y)
- Federal Funds Rate Range

Black Diamond Portfolio Performance Report - Two Separated Tables (MUST ALWAYS BE TABLES, WITHOUT EXCEPTION):

- Portfolio Allocation Table (from Black Diamond, with percentages).
 STRICT RULES:
 • You MUST ONLY extract this from the Black Diamond context.
```

```
 • You MUST ALWAYS show ALL asset classes EXACTLY as they appear in Black Diamond, including smaller categories such as International Bonds, Emerging Markets, Alternatives, or any other.
 • You MUST NEVER omit, summarize, merge, collapse, or rename any asset class. Even if an asset class is very small (e.g., <1%), it MUST still be shown as its own row with its exact percentage.
 • Asset class names MUST be copied VERBATIM from Black Diamond.
 • Percentages MUST be copied VERBATIM from Black Diamond.
 • Even if these values are referenced earlier (e.g., inside Investment Planning), you MUST still generate this table here again.

- Account-Level Returns Table (with percentages). REMEMBER THAT THIS BELONGS TO BLACK DIAMOND, AND NEVER MIX THIS WITH THE ACCOUNT BALANCES
 STRICT RULE: This table is mandatory. You MUST generate it here,
 always separated from the Portfolio Allocation table.

### **Accounts**
* Current cash needs and income sources:

YOU MUST INCLUDE the account balances in addition to the overall data here FOR ALL THE ACCOUNTS (organized by client name, account number, account type and model type per account). DO NOT IGNORE ANY ACCOUNT. INCLUDE ALL THE ACCOUNTS. YOU MUST ALWAYS PRESENT THE OWNER'S NAMES OF EACH ACCOUNT. When you present them in a table, put all of those aligned to the right to easily differentiate larger and shorter account balances, so that visually all decimal points can be in the same position. Include the Total Account Balances row inside the same account balances table, properly formatted as the final row, with the label spanning the appropriate columns and the total amount right-aligned in the balance column.
- The accounts must always be presented as a table.
- This table must always be present.
- The table must always include the following columns:
 - Account Number
 - Account Owner
 - Account Type
 - Model Type
 - Balance
- The Total Account Balances row must always be present in the end of the table.

### Transactions - Two Mandatory Tables (MUST ALWAYS BE PRESENT, WITHOUT EXCEPTION):

- Both tables must always include: CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU MUST NOT IGNORE ANY OF THEM
```

- You must always generate two tables: Transactions (YTD) and Transactions (Prior Year), never forget to include them.
- The two tables must always contain (In each of them, for all the existent accounts):
 - Account Number
 - Contributions
 - Withdrawals
 - Rollovers
 - Conversions
 - Federal Tax Withholdings
 - State Tax Withholdings

* Transactions (YTD) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD PRESENT ALL VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A TABLE.
* Transactions (Prior Year) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD PRESENT ALL VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A TABLE.

Net Worth: YOU MUST EXTRACT THESE VALUES ONLY FROM RIGHTCAPITAL. IF NO VALUE EXIST IN THE RIGHTCAPITAL CONTEXT, YOU MUST OMIT THIS SECTION AND NOT INCLUDE ANY VALUE HERE. YOU SHOULD PRESENT THIS AS A TABLE
- Assets
- Liabilities
- Net Worth

Beneficiaries (Table(s)) - These must be tables based upon each existent client:
- The table(s) MUST include EXACTLY these columns (in this order):
 • Name
 • Type (Primary / Contingent)
 • % of Shares
- Beneficiary Type MUST be normalized to either "Primary" or "Contingent" only.
- % of Shares MUST be shown as a percentage with the % symbol (e.g., 50%).
"""

 # => Client Profile Sections (Include the sections aligned with the client profile) - EACH SECTION'S CONTENT MUST BE DEVELOPED AS BULLET POINTS. ALL HERE ARE POTENTIAL SECTIONS, SO INFO. SHOULD BE INCLUDED AND NOT REPEAT THE SAME NAME OF THE SECTION WITH NO INFO. IN THE END OF EACH SECTION, YOU MUST INCLUDE THE DATA SOURCE WHERE YOU GOT THE INFORMATION FROM.

```
# - **Key focus:** Maximizing retirement readiness, preparing for transition

# - **Retirement Readiness**
#   - Projecting retirement income needs
#   - Gap analysis & accelerated savings

# - **Utilizing Proper Savings Strategies**
#   - Catch-Up Contributions
#   - Taking advantage of 50+ catch-up limits for Retirement Accounts (IRA, 401(k), 403(b), 457(b), etc.) and HSA

# - **Pension | Social Security | Benefit Decisions**
#   - Early evaluation of pension payout options
#   - Social Security claiming strategies (start planning before 62)

# - **Tax Planning for Retirement**
#   - Managing Tax Brackets
#   - Pre – Tax vs After Tax

# - **Healthcare Transition**
#   - Preparing for Medicare eligibility at 65 and bridging the gap between retirement and Medicare age
#   - Evaluating timing of Retirement – Exchange vs ACA vs working part time for benefits

# - **Estate Planning Enhancement**
#   - Updating wills, trusts, beneficiary designations
#   - Charitable giving strategies

# - **Risk Management**
#   - Portfolio Transition
#   - Shifting allocation to be more balanced as retirement approaches
#   - Introducing income-focused investments

# - **Taxes**
#   - Taxable Income from previous year
#   - Federal
#   - State
#   - Tax Bracket

# - **Estate Planning (Documents on file)**
#   - Wills
```

```
#   - Healthcare POAs
#   - Durable POAs
#   - Revocable Trust
#   - Beneficiaries

# - **Cash Flow**
#   - Review Variables
#   - Savings for Retirement and Kids (If they have kids)
#   - Retirement Accounts
#   - 529's / UTMA's (If they have kids)

# - **Retirement Projections**
#   - Projected Retirement Dates
#   - Review Projections
#   - Rate of Return - Conservative
#   - Rate of Return - Aggressive
#   - Timeline & trend

# ---


FROM_62_AND_UP_AGENDA_TEMPLATE = f"""
### **2025 Overall Planning**

#### Last Meeting - Topics & Client Updates
- You must list all the topics that were discussed in the last meeting here. ALWAYS MUST BE WRITTEN IN PAST.
- You must also list all the client updates from the last meeting here. These updates may be sentences or questions, but they must always relate to the client's profile and personal activities (distinct from the meeting topics). ALWAYS MUST BE WRITTEN IN PAST.

#### Topics that we will discuss in this meeting - YOU MUST ALWAYS PUT THIS SECTION:
- You must list all the topics that we will discuss in the meeting here.

* "Any changes since we last met?" ← Always put this
* A placeholder for open-ended or major life event discussions:

Client(s) Age(s): The age(s) of the client(s) must be included here.

{DOMAIN_SPECIFIC_SECTIONS}
```

```
Market Update:
The Market Updates should go here (AS A TABLE):
- US Total Market (ITOT)
- International Total Market (IXUS)
- Total Bond Market (AGG)
- US Small Cap (IJR)
- US Mid Cap (IJH)
- US Ten Year Yield (US10Y)
- Federal Funds Rate Range

Black Diamond Portfolio Performance Report - Two Separated Tables (MUST ALWAYS BE TABLES, WITHOUT EXCEPTION):

- Portfolio Allocation Table (from Black Diamond, with percentages).
 STRICT RULES:
 • You MUST ONLY extract this from the Black Diamond context.
 • You MUST ALWAYS show ALL asset classes EXACTLY as they appear in Black Diamond, including smaller categories such as International Bonds, Emerging Markets, Alternatives, or any other.
 • You MUST NEVER omit, summarize, merge, collapse, or rename any asset class. Even if an asset class is very small (e.g., <1%), it MUST still be shown as its own row with its exact percentage.
 • Asset class names MUST be copied VERBATIM from Black Diamond.
 • Percentages MUST be copied VERBATIM from Black Diamond.
 • Even if these values are referenced earlier (e.g., inside Investment Planning), you MUST still generate this table here again.

- Account-Level Returns Table (with percentages). REMEMBER THAT THIS BELONGS TO BLACK DIAMOND, AND NEVER MIX THIS WITH THE ACCOUNT BALANCES
 STRICT RULE: This table is mandatory. You MUST generate it here,
 always separated from the Portfolio Allocation table.

### **Accounts**
* Current cash needs and income sources:

YOU MUST INCLUDE the account balances in addition to the overall data here FOR ALL THE ACCOUNTS (organized by client name, account number, account type and model type per account). DO NOT IGNORE ANY ACCOUNT. INCLUDE ALL THE ACCOUNTS. YOU MUST ALWAYS PRESENT THE OWNER'S NAMES OF EACH ACCOUNT. When you present them in a table, put all of those aligned to the right to easily differentiate larger and shorter account balances, so that visually all decimal points can be in the same position. Include the Total Account Balances row inside the same account balances table, properly formatted as the
```

```
final row, with the label spanning the appropriate columns and the total amount
right-aligned in the balance column.
- The accounts must always be presented as a table.
- This table must always be present.
- The table must always include the following columns:
 - Account Number
 - Account Owner
 - Account Type
 - Model Type
 - Balance
- The Total Account Balances row must always be present in the end of the table.

### Transactions - Two Mandatory Tables (MUST ALWAYS BE PRESENT, WITHOUT EXCEPTION):

- Both tables must always include: CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS, CONVERSIONS,
FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU MUST NOT IGNORE ANY OF THEM
- You must always generate two tables: Transactions (YTD) and Transactions (Prior
Year), never forget to include them.
- The two tables must always contain (In each of them, for all the existent accounts):
 - Account Number
 - Contributions
 - Withdrawals
 - Rollovers
 - Conversions
 - Federal Tax Withholdings
 - State Tax Withholdings

* Transactions (YTD) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS, ROLLOVERS,
CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD PRESENT ALL
VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A TABLE.
* Transactions (Prior Year) - YOU MUST INCLUDE ALL CONTRIBUTIONS, WITHDRAWALS,
ROLLOVERS, CONVERSIONS, FEDERAL TAX WITHHOLDING, STATE TAX WITHHOLDING. YOU SHOULD
PRESENT ALL VALUES, EVEN THOUGH SOME ARE 0. YOU SHOULD PRESENT THESE VALUES AS A
TABLE.

### Net Worth: YOU MUST EXTRACT THESE VALUES ONLY FROM RIGHTCAPITAL. IF NO VALUE EXIST
IN THE RIGHTCAPITAL CONTEXT, YOU MUST OMIT THIS SECTION AND NOT INCLUDE ANY VALUE
HERE. YOU SHOULD PRESENT THIS AS A TABLE
- Assets
- Liabilities
- Net Worth
```

```
- **Required Minimum Distributions (RMDs)**
 - Estimated RMDs from 62-72 to plan tax strategy early (if applicable) - List the Anticipated RMDs here if exists. YOU MUST EXTRACT THIS DATA FROM THE CUSTODIANS, NOT OTHER CONTEXT. Don't put "Left to Satisfy" field in the Anticipated RMDs table.
 - Existent RMDs from 73 and up (if applicable) - List the RMDs here if exists. YOU MUST EXTRACT THIS DATA FROM THE CUSTODIANS, NOT OTHER CONTEXT. In this table, you must present the "Left to Satisfy" field as how it is defined, don't create a new column for 'Year-To-Date Distribution'. Don't present the Tax Impact field. NEVER IGNORE THE LEFT TO SATISFY AMOUNT.
 - Roth conversions before RMD age
   - Paying the Tax Now vs Later
   - Tax Free Growth for you and beneficiaries
   - Lower Tax Bracket

- Qualified Charitable Distributions (QCDs)
 - YOU MUST NOT IGNORE THIS IF QCDs ARE PRESENT.
 - List ALL the QCDs here as a table. YOU MUST EXTRACT THIS DATA FROM THE CUSTODIANS, NOT OTHER CONTEXT.

### Beneficiaries (Table(s)) - These must be tables based upon each existent client:
- The table(s) MUST include EXACTLY these columns (in this order):
 • Name
 • Type (Primary / Contingent)
 • % of Shares
- Beneficiary Type MUST be normalized to either "Primary" or "Contingent" only.
- % of Shares MUST be shown as a percentage with the % symbol (e.g., 50%).
"""



# => Client Profile Sections (Include the sections aligned with the client profile) - EACH SECTION'S CONTENT MUST BE DEVELOPED AS BULLET POINTS. ALL HERE ARE POTENTIAL SECTIONS, SO INFO. SHOULD BE INCLUDED AND NOT REPEAT THE SAME NAME OF THE SECTION WITH NO INFO. IN THE END OF EACH SECTION, YOU MUST INCLUDE THE DATA SOURCE WHERE YOU GOT THE INFORMATION FROM.

# - **Key focus:** Income, tax-efficient withdrawals, longevity risk management

# - **Social Security Optimization**
#   - Claiming strategy discussions (62, FRA, or 70)
#     - Running a Breakeven based on numbers
#     - 8% growth each year you delay it
```

```
# - **Medicare Planning**
#   - Enrollment deadlines & coverage choices
#   - IRMAA impact & strategies to reduce surcharges

# - **Withdrawal Strategies**
#   - Tax-efficient income sequencing (taxable → tax-deferred → Roth)
#   - Managing capital gains in retirement

# - **Longevity & Legacy Planning**
#   - Sustainable withdrawal rates
#   - Legacy goals and charitable giving

# - **Risk Management**
#   - Managing healthcare and other medical costs
#   - Portfolio adjustments as needed

# - **Estate & Gifting**
#   - Annual gifting strategies
#   - Trust updates to reflect current goals

# - **Lifestyle Planning**
#   - Housing downsizing or relocation
#   - Funding travel/hobbies while making sure we are not spending down the portfolio

# - **Taxes**
#   - Taxable Income from previous year
#   - Federal
#   - State
#   - Tax Bracket

# - **Estate Planning (Documents on file)**
#   - Wills
#   - Healthcare POAs
#   - Durable POAs
#   - Revocable Trust
#   - Beneficiaries (LIST ALL THE BENEFICIARIES)

# - **Social Security Benefits**
#   - Person (if applicable)
#   - Person (if applicable)

# ---
```

```
# ### **To-Do List / Action Items / Next Steps / Concerns** => SHOW THIS ALWAYS IN THE END

# * Forms to sign, follow-ups, document requests
# * Upcoming deadlines
# * Plan for the next meeting
# ---
```

```
       if client_age_number <= 35:
           additional_topics = f"""
- **Key focus:** Building habits, establishing financial security, understanding time is on your side
```

```
- Fixed Income Positions (YOU MUST USE ALL OF THEM AS TABLES FOR EACH ACCOUNT AND SPECIFY THE FIXED INCOME POSITION TYPE TOO, DON'T SUMMARIZE OR IGNORE ANY OF THESE): {input.get('custodians_data', {}).get('positions', '')}
   - Remember that FIXED INCOME POSITIONS MUST ALWAYS BE PRESENTED AS TABES, NEVER AS LISTS
   - Do NOT create a section named 'Portfolio Updates' or any generic 'Updates' section. Start directly with Fixed Income Positions when positions exist
   - Always leave ONE blank line before and after each table to prevent Markdown rendering issues
```

```
- **Budgeting**
 - Student loan repayment strategies
 - Credit score building/repair
 - Cash Flow Planning - What is coming in vs going out
 - Emergency fund targets (3-6 months)
```

```
- **Retirement Savings - Establishing a good starting point**
 - Getting started early
   - Match from your 401k
   - Maxing your Roth
   - Allocating any extra cash in general
 - Utilizing compounding growth
```

```
- **Career & Income Growth**
 - Understanding Career Path
 - Benefits analysis
```

```
- **Insurance Basics - Income Replacement and the fact you will never be healthier than you are now**
 - Term life insurance to supplant debts and take care of dependents if the worst happens
 - Understanding how much coverage is enough
 - Taking advantage of what employer offers
```

```
- **Goals - Time Horizon and Risk Tolerance**
 - Buying a first home
 - Saving for marriage/kids
 - Setting yourself up for the future
 - Not rushing into anything unless it makes sense for you
```

```
- **Tax Planning**
 - Maximizing deductions/credits
 - Pre-Tax vs After-Tax
```

```
- **Investing Basics**
 - Risk tolerance
 - Building diversified, long-term portfolios
 - Not seeking a one-hit wonder
```

```
- **Taxes**
 - Taxable Income (2024)
 - Federal
 - State
 - Bracket
```

```
- **Top Financial Goals**
 - Needs
 - Wants
 - Wishes
```

```
- **Income Planning**
 - Husband
 - Wife
 - Other Income
 - Total Income
```

```
- **Insurance Planning**
 - Husband
 - Wife
```

```
- **Employer Review**
 - Retirement Accounts
 - % of Contributions
 - Review Allocation
 - Benefits
 - Career Path
```

```
---
          """
      elif client_age_number >= 36 and client_age_number <= 49:
          additional_topics = f"""
- **Key focus:** Growing assets, being tax efficient, protecting against risks
```

```
- Fixed Income Positions (YOU MUST USE ALL OF THEM AS TABLES FOR EACH ACCOUNT AND SPECIFY THE FIXED INCOME POSITION TYPE TOO, DON'T SUMMARIZE OR IGNORE ANY OF THESE): {input.get('custodians_data', {}).get('positions', '')}
   - Remember that FIXED INCOME POSITIONS MUST ALWAYS BE PRESENTED AS TABES, NEVER AS LISTS
   - Do NOT create a section named 'Portfolio Updates' or any generic 'Updates' section. Start directly with Fixed Income Positions when positions exist
   - Always leave ONE blank line before and after each table to prevent Markdown rendering issues
```

```
- **Retirement Projections - Start getting a feeling for where you are in life**
 - Increasing contributions toward 401(k)/IRA
 - Pre-Tax vs After-Tax
 - Catch-up strategies if behind
```

```
- **College Planning**
 - 529 plan funding strategies
 - Balancing retirement vs. education savings - how much is enough
```

```
- **Insurance Review**
 - As you enter your prime working years - make sure you are properly insured for the people around you
 - Additional insurance options
   - Long-term care
   - Disability
```

```
- **Cash Flow Strategies**
 - Mortgage payoff strategies
 - Avoiding living outside of your means
 - Strategizing which is good vs. bad debt
```

```
- **Tax Efficiency**
 - Maximizing employer benefits (FSA, HSA)
```

```
- **Estate Planning**
 - Wills, powers of attorney, guardianship for minors
```

```
- **Investment Allocation Review**
 - Shifting from aggressive growth to balanced risk tolerance as goals become clearer and change
```

```
- **Business/Side Income**
 - Retirement planning for self-employed
 - Business planning
```

```
- **Review Variables**
 - **Savings for Retirement and Kids**
   - Retirement Accounts
   - 529's / UTMA's
```

```
- **Taxes**
 - Taxable Income (2024)
```

```
 - Federal
 - State
 - Tax Bracket
```

```
- **Income Planning**
 - Husband
 - Wife
 - Total Income
```

```
- **Insurance Planning**
 - Husband
 - Wife
```

```
- **Employer Review**
 - Retirement Accounts
 - % of Contributions
 - Review Allocation
 - Benefits
 - Career Path
```

```
- **Estate Planning (Documents on file)**
 - Wills
 - Healthcare POAs
 - Durable POAs
 - Revocable Trust
 - Beneficiaries (LIST ALL THE BENEFICIARIES)
```

```
- **Retirement Projections**
 - Projected Retirement Dates
 - Review Projections
 - Rate of Return – Conservative
 - Rate of Return – Aggressive
 - Timeline & trend

---
           """
       elif client_age_number >= 50 and client_age_number <= 61:
           additional_topics = f"""
- **Key focus:** Maximizing retirement readiness, preparing for transition
```

```
- Fixed Income Positions (YOU MUST USE ALL OF THEM AS TABLES FOR EACH ACCOUNT AND
SPECIFY THE FIXED INCOME POSITION TYPE TOO, DON'T SUMMARIZE OR IGNORE ANY OF THESE):
{input.get('custodians_data', {}).get('positions', '')}
   - Remember that FIXED INCOME POSITIONS MUST ALWAYS BE PRESENTED AS TABES, NEVER AS
LISTS
   - Do NOT create a section named 'Portfolio Updates' or any generic 'Updates'
section. Start directly with Fixed Income Positions when positions exist
   - Always leave ONE blank line before and after each table to prevent Markdown
rendering issues
```

```
- **Retirement Readiness**
 - Projecting retirement income needs
 - Gap analysis & accelerated savings
```

```
- **Utilizing Proper Savings Strategies**
 - Catch-Up Contributions
 - Taking advantage of 50+ catch-up limits for 401(k)/IRA/HSA
```

```
- **Pension | Social Security | Benefit Decisions**
 - Early evaluation of pension payout options
 - Social Security claiming strategies (start planning before 62)
```

```
- **Tax Planning for Retirement**
 - Managing Tax Brackets
 - Pre – Tax vs After Tax
```

```
- **Healthcare Transition**
 - Preparing for Medicare eligibility at 65 and bridging the gap between retirement
and Medicare age
 - Evaluating timing of Retirement – Exchange vs ACA vs working part time for benefits
```

```
- **Estate Planning Enhancement**
 - Updating wills, trusts, beneficiary designations
 - Charitable giving strategies
```

```
- **Risk Management**
 - Portfolio Transition
 - Shifting allocation to be more balanced as retirement approaches
 - Introducing income-focused investments
```

```
- **Taxes**
 - Taxable Income from previous year
 - Federal
 - State
 - Tax Bracket
```

```
- **Estate Planning (Documents on file)**
 - Wills
 - Healthcare POAs
 - Durable POAs
 - Revocable Trust
 - Beneficiaries
```

```
- **Cash Flow**
 - Review Variables
 - Savings for Retirement and Kids (If they have kids)
 - Retirement Accounts
 - 529’s / UTMA’s (If they have kids)
```

```
- **Retirement Projections**
 - Projected Retirement Dates
 - Review Projections
 - Rate of Return – Conservative
 - Rate of Return – Aggressive
 - Timeline & trend
```

```
---
          """
      elif client_age_number >= 62:
          additional_topics = f"""
- **Key focus:** Income, tax-efficient withdrawals, longevity risk management
```

```
- Fixed Income Positions (YOU MUST USE ALL OF THEM AS TABLES FOR EACH ACCOUNT AND
SPECIFY THE FIXED INCOME POSITION TYPE TOO, DON'T SUMMARIZE OR IGNORE ANY OF THESE):
{input.get('custodians_data', {}).get('positions', '')}
   - Remember that FIXED INCOME POSITIONS MUST ALWAYS BE PRESENTED AS TABES, NEVER AS
LISTS
   - Do NOT create a section named 'Portfolio Updates' or any generic 'Updates'
section. Start directly with Fixed Income Positions when positions exist
   - Always leave ONE blank line before and after each table to prevent Markdown
rendering issues
```

```
- **Social Security Optimization**
 - Claiming strategy discussions (62, FRA, or 70)
   - Running a Breakeven based on numbers
   - 8% growth each year you delay it
```

```
- **Medicare Planning**
 - Enrollment deadlines & coverage choices
 - IRMAA impact & strategies to reduce surcharges
```

```
- **Withdrawal Strategies**
 - Tax-efficient income sequencing (taxable → tax-deferred → Roth)
 - Managing capital gains in retirement
```

```
- **Required Minimum Distributions (RMDs)**
 - Estimated RMDs from 62-72 to plan tax strategy early (if applicable) - List the
Anticipated RMDs here if exists. YOU MUST EXTRACT THIS DATA FROM THE CUSTODIANS, NOT
OTHER CONTEXT. Don't put "Left to Satisfy" field in the Anticipated RMDs table.
 - Existent RMDs from 73 and up (if applicable) - List the RMDs here if exists. YOU
MUST EXTRACT THIS DATA FROM THE CUSTODIANS, NOT OTHER CONTEXT. In this table, you must
present the "Left to Satisfy" field as how it is defined, don't create a new column
for 'Year-To-Date Distribution'. Don't present the Tax Impact field. NEVER IGNORE THE
LEFT TO SATISFY AMOUNT.
 - Roth conversions before RMD age
   - Paying the Tax Now vs Later
   - Tax Free Growth for you and beneficiaries
   - Lower Tax Bracket
```

```
- Qualified Charitable Distributions (QCDs)
 - List the QCDs here if exists. YOU MUST EXTRACT THIS DATA FROM THE CUSTODIANS, NOT
OTHER CONTEXT.
```

```
- **Longevity & Legacy Planning**
 - Sustainable withdrawal rates
 - Legacy goals and charitable giving
```

```
- **Risk Management**
```

```
 - Managing healthcare and other medical costs
 - Portfolio adjustments as needed
```

```
- **Estate & Gifting**
 - Annual gifting strategies
 - Trust updates to reflect current goals
```

```
- **Lifestyle Planning**
 - Housing downsizing or relocation
 - Funding travel/hobbies while making sure we are not spending down the portfolio
```

```
- **Taxes**
 - Taxable Income from previous year
 - Federal
 - State
 - Tax Bracket
```

```
- **Estate Planning (Documents on file)**
 - Wills
 - Healthcare POAs
 - Durable POAs
 - Revocable Trust
 - Beneficiaries (LIST ALL THE BENEFICIARIES)
```

```
- **Social Security Benefits**
 - Person (if applicable)
 - Person (if applicable)
```

```
---
          """
```

```
def generate_agenda_v2(
   input: AgendaCompilationV2Request
):
   """
   This function is used to build the agenda generator graph.
   """
   try:
       import psutil
       import time
       import os
       import platform
       import uuid
       from datetime import datetime
       from api.app.helpers.agenda_audit_metadata_manager import
AgendaAuditMetadataManager
       from api.app.helpers.azure_postgresql_manager import LookupClientAdvisorService
```

```
      send_agenda_compilation_started_notification(input.client_name,
"vantage-financial-agenda-generation-notifications")

      # Initialize performance tracking
      start_time = time.time()
      process = psutil.Process()
      initial_memory = process.memory_info().rss / 1024 / 1024  # MB

      # Initialize metadata dict
      metadata = {
          'start_time': datetime.now().isoformat(),
          'node_execution_times': {},
          'node_execution_order': [],
          'memory_usage': {},
          'errors': [],
      }

      lookup_household = HouseholdLookupService()
      household_result = lookup_household.get_latest_record(input.client_name)

      custodian_id = int(household_result['provider_id'])

      agenda_trace_id = f"agenda_generation_{input.client_name}_{int(time.time())}"
      trace_id = str(uuid.uuid4())

      # Wrap each node function to track performance
      def wrap_node(node_func, node_name):
          def wrapped(*args, **kwargs):
              node_start = time.time()
              metadata['node_execution_order'].append(node_name)

              try:
                  # Track memory before node execution
                  mem_before = process.memory_info().rss / 1024 / 1024  # MB

                  # Execute node
                  result = node_func(*args, **kwargs)
                  # Track performance metrics
                  execution_time = time.time() - node_start
                  mem_after = process.memory_info().rss / 1024 / 1024  # MB
```

```
                    metadata['node_execution_times'][node_name] = execution_time
                    metadata['memory_usage'][node_name] = {
                        'before': mem_before,
                        'after': mem_after,
                        'diff': mem_after - mem_before
                    }

                    return result
                except Exception as e:
                    metadata['errors'].append({
                        'node': node_name,
                        'error': str(e),
                        'timestamp': datetime.now().isoformat()
                    })
                    raise
            return wrapped

        graph_builder = StateGraph(AgendaGeneratorGraph)

        # Add nodes with performance tracking
        graph_builder.add_node("redtail_graph_node", wrap_node(redtail_graph,
"redtail_graph_node"))
        graph_builder.add_node("custodians_graph_node", wrap_node(custodians_graph,
"custodians_graph_node"))
        graph_builder.add_node("planning_and_retirement_graph_node",
wrap_node(planning_and_retirement_graph, "planning_and_retirement_graph_node"))
        graph_builder.add_node("portfolio_risk_review_graph_node",
wrap_node(portfolio_risk_review_graph, "portfolio_risk_review_graph_node"))
        graph_builder.add_node("estate_insurance_government_graph_node",
wrap_node(estate_insurance_government_graph,
"estate_insurance_government_graph_node"))
        graph_builder.add_node("generate_main_agenda_node",
wrap_node(generate_main_agenda, "generate_main_agenda_node"), defer=True)
        graph_builder.add_node("generate_visual_agenda_node",
wrap_node(generate_visual_agenda_v2, "generate_visual_agenda_node"))

        graph_builder.add_edge(START, "redtail_graph_node")
        graph_builder.add_edge("redtail_graph_node", "custodians_graph_node")
        graph_builder.add_edge("redtail_graph_node",
"planning_and_retirement_graph_node")
        graph_builder.add_edge("redtail_graph_node",
"portfolio_risk_review_graph_node")
```

```
        graph_builder.add_edge("planning_and_retirement_graph_node",
"estate_insurance_government_graph_node")

        graph_builder.add_edge("custodians_graph_node", "generate_main_agenda_node")
        graph_builder.add_edge("portfolio_risk_review_graph_node",
"generate_main_agenda_node")
        graph_builder.add_edge("estate_insurance_government_graph_node",
"generate_main_agenda_node")

        graph_builder.add_edge("generate_main_agenda_node",
"generate_visual_agenda_node")
        graph_builder.add_edge("generate_visual_agenda_node", END)
```