Sudhakar_TFinalProject

October 21, 2025

1 Multi-Agent Financial Analysis System

Github Link: https://github.com/ApexZenT/Course_8_NLP/tree/master/src/Week7/Multi-Agent%20Financial%20Analysis%20System/FinalProject

[10]:
```
!pip install gpt4all
```

```
Requirement already satisfied: gpt4all in /usr/local/lib/python3.12/dist-
packages (2.8.2)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-
packages (from gpt4all) (2.32.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages
(from gpt4all) (4.67.1)
Requirement already satisfied: charset_normalizer<4,>=2 in
/usr/local/lib/python3.12/dist-packages (from requests->gpt4all) (3.4.4)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-
packages (from requests->gpt4all) (3.11)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.12/dist-packages (from requests->gpt4all) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.12/dist-packages (from requests->gpt4all) (2025.10.5)
```

[11]:
```
# --- Standard Library ---
import logging
import json
from pathlib import Path
from datetime import datetime, timezone
from dateutil import parser
from typing import List, Optional

# --- Third-party Libraries ---
import requests
import polars as pl
import yfinance as yf
from dataclasses import dataclass
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy import create_engine, Column, Integer, String, Text, DateTime
from sqlalchemy.orm import declarative_base, sessionmaker
from gpt4all import GPT4All

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
```

[12]:
```
# Global settings for the Multi-Agent Financial Analysis System


# Set this flag to True after processing CSV and ingesting into DB to avoid␣
↪re-processing
load_csv_to_db='false'

# News API configuration
news_api_key='NEWS_API_KEY'
news_api_endpoint_everything='https://newsapi.org/v2/everything'
news_api_endpoint_top_headlines='https://newsapi.org/v2/top-headlines'

# FRED API Configuration
fred_api_key='FRES_API_KEY'
fred_api_endpoint='https://api.stlouisfed.org/fred'

# DB path
news_db_path='/data/processed/news.db'

# default limit for fetching news records
default_news_fetch_limit='1000'
```

[13]:
```
# Paths
BASE_PATH = Path('./data/processed')
BASE_PATH.mkdir(parents=True, exist_ok=True)

# --- News DB ---
NEWS_DATABASE_URL = f"sqlite:///{BASE_PATH}/news.db"
engine_news = create_engine(NEWS_DATABASE_URL, echo=False,␣
↪connect_args={"check_same_thread": False})
SessionNews = sessionmaker(bind=engine_news, expire_on_commit=False,␣
↪autoflush=False)
BaseNews = declarative_base()

# --- Agent Memory DB ---
AGENTMEMORY_DATABASE_URL = f"sqlite:///{BASE_PATH}/agent_memory.db"
engine_agent_memory = create_engine(AGENTMEMORY_DATABASE_URL, echo=False,␣
↪connect_args={"check_same_thread": False})
SessionAgentMemory = sessionmaker(bind=engine_agent_memory,␣
↪expire_on_commit=False, autoflush=False)
BaseAgentMemory = declarative_base()
```

```
# Initialize tables
def init_db():
    BaseNews.metadata.create_all(bind=engine_news)
    BaseAgentMemory.metadata.create_all(bind=engine_agent_memory)
    logger.info("Database tables created successfully")

init_db()  # Call once
```

[14]:
```
# --- ORM Models ---
class News(BaseNews):
    __tablename__ = 'news'
    __table_args__ = {'extend_existing': True}
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String, index=True)
    description = Column(String)
    source = Column(String, index=True)
    url = Column(String, unique=True, index=True)
    publishedAt = Column(DateTime, index=True)

class AgentMemory(BaseAgentMemory):
    __tablename__ = 'agent_memory'
    __table_args__ = {'extend_existing': True}
    id = Column(Integer, primary_key=True)
    agent_name = Column(String)
    context = Column(String)
    inputs = Column(Text)
    output = Column(Text)
    timestamp = Column(String)

# --- DTOs ---

@dataclass
class NewsDTO:
    title: str
    publishedAt: Optional[datetime] = None
    description: Optional[str] = None
    source: Optional[str] = None
    url: Optional[str] = None

@dataclass
class AgentMemoryDTO:
    agent_name: str
    context: str
    inputs: str
    output: str
    timestamp: Optional[str] = None
```

[15]:
```python
class CSVAdapter:
    """
    Adapter for loading and normalizing financial news CSVs
    (e.g., CNBC, Reuters, Bloomberg) into a common schema.
    """

    BASE_PATH = Path('./data/raw')

    _rename_map = {
        "headline": "title",
        "headlines": "title",
        "content": "description",
        "summary_text": "description",
        "summary": "description",
        "date_published": "publishedAt",
        "timestamp": "publishedAt",
        "time": "publishedAt",
    }

    def fetch_news_from_csv(self) -> List[NewsDTO]:
        all_news: List[NewsDTO] = []

        csv_files = list(self.BASE_PATH.glob("*.csv"))
        if not csv_files:
            logger.warning("No CSV files found in %s", self.BASE_PATH)
            return all_news

        logger.info("Found %d CSV files to process.", len(csv_files))

        for csv_file in csv_files:
            try:
                df = pl.read_csv(csv_file)

                # Standardize column names
                df = df.rename(
                    {
                        col: col.replace("-", "_").replace(" ", "_").lower().
↪strip()
                        for col in df.columns
                    }
                )
                df = df.rename(self._rename_map, strict=False)

                filtered_df = df.filter(
                    pl.col("title").is_not_null() & (pl.col("title") != "")
                )
                rows_as_dicts = filtered_df.to_dicts()
                logger.info(
                    "Processing %d rows from CSV file: %s",
                    len(rows_as_dicts),
                    csv_file.name,
                )

                for row in rows_as_dicts:
                    news_item = NewsDTO(
                        title=row.get("title"),
                        publishedAt=parse_datetime(row.get("publishedAt")),
                        description=row.get("description"),
                        source=csv_file.stem,  # use file name as source
                        url=row.get("url"),
                    )
                    all_news.append(news_item)

            except Exception as e:
                logger.error("Error processing CSV file %s: %s", csv_file.name,␣
↪e)

        logger.info("Total news items fetched from CSVs: %d", len(all_news))
        return all_news


class NewsAdapter:
    """Adapter for fetching and normalizing financial news from external APIs.
↪"""

    def __init__(self, api_key: str):
        self.api_key = api_key
        logger.info("Initialized NewsAdapter with provided API key.")

    def fetch_news(self, endpoint: str, params: dict) -> List[NewsDTO]:
        """Fetch news from an API endpoint and return as list of NewsDTOs."""
        params["apiKey"] = self.api_key
        try:
            response = requests.get(endpoint, params=params)
            response.raise_for_status()
            articles = self._parse_articles(response.json())
            logger.info("Fetched %d articles from endpoint %s", len(articles),␣
↪endpoint)
            return articles
        except requests.RequestException as e:
            logger.error("Request failed for endpoint %s: %s", endpoint, e)
            return []
        except Exception as e:
            logger.error("Unexpected error while fetching news: %s", e)
            return []

    def _parse_articles(self, response_json: dict) -> List[NewsDTO]:
        """Parse the JSON response into a list of NewsDTO objects."""
        articles = response_json.get("articles", [])
        news_list = []
        for a in articles:
            try:
                news_item = NewsDTO(
                    title=a.get("title"),
                    description=a.get("description"),
                    source=a.get("source", {}).get("name"),
                    publishedAt=parse_datetime(a.get("publishedAt")),
                    url=a.get("url"),
                )
                news_list.append(news_item)
            except Exception as e:
                logger.warning("Failed to parse article: %s", e)
        return news_list
```

[16]:
```
# Utilities
def save_to_db(session_factory: sessionmaker, orm_instance) -> bool:
    """
    Generic utility to save a pre-built ORM instance to the database.

    Args:
        session_factory: SQLAlchemy sessionmaker
        orm_instance: An instance of a SQLAlchemy ORM class

    Returns:
        bool: True if saved successfully, False otherwise
    """
    with session_factory() as session:
        try:
            session.add(orm_instance)
            session.commit()
            logger.info("Saved ORM instance: %s", orm_instance)
            return True
        except SQLAlchemyError as e:
            session.rollback()
            logger.error("Failed to save ORM instance: %s | Error: %s",␣
 ↪orm_instance, e)
            return False


# Map common timezone abbreviations to UTC offsets (seconds)
```

```
TZINFOS = {
    "ET": -5 * 3600,
    "EST": -5 * 3600,
    "EDT": -4 * 3600,
    "CT": -6 * 3600,
    "CST": -6 * 3600,
    "CDT": -5 * 3600,
    "MT": -7 * 3600,
    "MST": -7 * 3600,
    "MDT": -6 * 3600,
    "PT": -8 * 3600,
    "PST": -8 * 3600,
    "PDT": -7 * 3600,
}


def parse_datetime(
    dt_str: Optional[str], default_tz: timezone = timezone.utc
) -> Optional[datetime]:
    """
    Parse a datetime string into a Python datetime object.

    Args:
        dt_str: The string to parse.
        default_tz: Timezone to assign if the string is naive.

    Returns:
        datetime object (aware) or None if input is None or invalid.
    """
    if not dt_str:
        return None

    try:
        dt = parser.parse(dt_str, tzinfos=TZINFOS)
        if dt.tzinfo is None:
            dt = dt.replace(tzinfo=default_tz)
        # Normalize to UTC
        dt = dt.astimezone(timezone.utc)
        return dt
    except (ValueError, TypeError) as e:
        logger.warning("Could not parse datetime '%s': %s", dt_str, e)
        return None
```

[17]:
```
class NewsDB:
    """DAO / DAL for News table using SQLAlchemy ORM."""

    def __init__(self):
        self.session = SessionNews()

    def __enter__(self):
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.close()

    def insert_news(self, news_items: List[NewsDTO]):
        """Insert multiple news records into the database in batches, skipping␣
↪duplicates."""
        if not news_items:
            logger.info("No news items to process.")
            return

        titles_set = {item.title for item in news_items if item.title}
        titles_list = list(titles_set)

        # Check for existing titles in batches
        existing_titles = set()
        logger.info("Checking for existing titles in batches...")
        chunk_size_lookup = 500
        for i in range(0, len(titles_list), chunk_size_lookup):
            batch_titles = titles_list[i : i + chunk_size_lookup]
            existing_in_batch = (
                self.session.query(News.title)
                .filter(News.title.in_(batch_titles))
                .all()
            )
            existing_titles.update(title for (title,) in existing_in_batch)
        logger.info("Found %d existing articles.", len(existing_titles))

        # Prepare new records
        new_records = [
            {
                "title": item.title,
                "publishedAt": item.publishedAt,
                "description": item.description,
                "source": item.source,
                "url": item.url,
            }
            for item in news_items
            if item.title not in existing_titles
        ]

        if not new_records:
            logger.info("No new articles to insert.")
            return

        # Bulk insert in chunks
        chunk_size_insert = 500
        try:
            for i in range(0, len(new_records), chunk_size_insert):
                batch = new_records[i : i + chunk_size_insert]
                self.session.bulk_insert_mappings(News, batch)
                logger.info(
                    "Inserted batch %d with %d items.",
                    i // chunk_size_insert + 1,
                    len(batch),
                )
            self.session.commit()
            logger.info("All new news items inserted successfully.")
        except Exception as e:
            self.session.rollback()
            logger.error("Error during bulk insert: %s", e)
            raise

    def fetch_news(
        self, limit: int = 1000, source: Optional[str] = None, q: Optional[str]␣
↪= None
    ) -> List[NewsDTO]:
        """Fetch news records from the database."""
        query = self.session.query(News)
        if source:
            query = query.filter(News.source == source)
        if q:
            query = query.filter(News.title.ilike(f"%{q}%"))

        records = query.limit(limit).all()
        news_dtos = [
            NewsDTO(
                title=r.title,
                publishedAt=r.publishedAt,
                description=r.description,
                source=r.source,
                url=r.url,
            )
            for r in records
        ]
        logger.info("Fetched %d news records from DB.", len(news_dtos))
        return news_dtos

    def close(self):
        self.session.close()
        logger.info("Database session closed.")

class NewsService:
    """Handles all business logic related to fetching and storing news."""

    load_limit: int = int(default_news_fetch_limit)
    load_csv_flag: bool = (load_csv_to_db == True)

    def __init__(self, api_key: str):
        self.adapter = NewsAdapter(api_key)
        self.db = NewsDB()

        if self.load_csv_flag:
            logger.info("CSV-to-DB load enabled. Starting initial load...")
            self._load_csv_to_db()
        else:
            logger.info("CSV-to-DB load disabled. Skipping initial load.")

    def _load_csv_to_db(self):
        """Private method to load CSV into DB only once."""
        csv_adapter = CSVAdapter()
        news_records = csv_adapter.fetch_news_from_csv()
        self.db.insert_news(news_records)
        logger.info("Inserted %d CSV records into DB.", len(news_records))

    def fetch_and_store_news(self, endpoint: str, **kwargs) -> List[NewsDTO]:
        """Fetch news using flexible parameters and optionally store to DB."""
        logger.debug("Fetching news from endpoint: %s | Params: %s", endpoint,␣
↪kwargs)
        news_items = self.adapter.fetch_news(endpoint, kwargs)
        self.db.insert_news(news_items)
        logger.info("Fetched and stored %d news items.", len(news_items))
        return news_items

    def get_stored_news(self, **kwargs) -> List[NewsDTO]:
        """Fetch stored news with flexible filtering options."""
        limit = kwargs.get("limit", self.load_limit)
        logger.debug("Fetching %d stored news records from DB.", limit)
        news = self.db.fetch_news(limit=limit)
        logger.info("Retrieved %d stored news records.", len(news))
        return news
```

[18]:
```
class NewsTool:
    """Tool exposing financial news fetching & DB operations."""

    name = "NewsTool"
    description = "Fetch and store financial news via NewsAPI and local DB."
    # Load configuration once
    _api_key: str = news_api_key
    _endpoint_everything: str = news_api_endpoint_everything
    _endpoint_top_headlines: str = news_api_endpoint_top_headlines

    _service: NewsService = None  # lazy init

    @property
    def service(self):
        """Initialize NewsService lazily."""
        if self._service is None:
            if not self._api_key:
                logger.error("Missing News API key in global-settings.
↪properties")
                raise ValueError("Missing News API key in global-settings.
↪properties")
            logger.info("Initializing NewsService...")
            self._service = NewsService(api_key=self._api_key)
        return self._service

    def fetch_everything(self, **kwargs) -> List[NewsDTO]:
        logger.debug("Fetching everything news with params: %s", kwargs)
        news_items = self.service.fetch_and_store_news(
            self._endpoint_everything, **kwargs
        )
        logger.info("Fetched %d 'everything' news items", len(news_items))
        return news_items

    def fetch_top_headlines(self, **kwargs) -> List[NewsDTO]:
        logger.debug("Fetching top headlines with params: %s", kwargs)
        news_items = self.service.fetch_and_store_news(
            self._endpoint_top_headlines, **kwargs
        )
        logger.info("Fetched %d top headlines", len(news_items))
        return news_items

    def fetch_news(self, source: str = "recent", **kwargs) -> List[NewsDTO]:
        logger.info("Fetching news from source: '%s'", source)
        news: List[NewsDTO] = []

        if source == "recent":
            news.extend(self.fetch_top_headlines(**kwargs))
        elif source == "history":
            news.extend(self.fetch_everything(**kwargs))
            news.extend(self.service.get_stored_news(**kwargs))
        elif source == "db":
            news.extend(self.service.get_stored_news(**kwargs))
        else:
            logger.error("Invalid news source '%s'", source)
            raise ValueError(f"Invalid news source '{source}'")

        # Deduplicate by URL
        seen = set()
        unique_news = []
        for article in news:
            key = getattr(article, "url", None)
            if key and key not in seen:
                seen.add(key)
                unique_news.append(article)

        logger.info("Returning %d unique news articles", len(unique_news))
        return unique_news



class StockTool:
    """A tool for fetching stock data using yfinance."""

    name = "StockAPI"
    description = (
        "A tool for fetching stock data. "
        "Use this tool to get historical stock prices, current stock prices,␣
↪and other financial data."
    )

    def _get_ticker(self, symbol: str):
        """Helper method to validate and return a yfinance Ticker object."""
        try:
            ticker = yf.Ticker(symbol)
            if not ticker.info:
                raise ValueError(f"Ticker symbol '{symbol}' not found or has no␣
↪info.")
            return ticker
        except Exception as e:
            logger.error("Error fetching ticker '%s': %s", symbol, e)
            raise

    def fetch_historical_data(
        self, symbol: str, period: str = "1yr", interval: str = "1d"
    ) -> dict:
        """Fetch historical stock data for a given ticker symbol."""
        logger.info(
            "Fetching historical data for %s | period=%s, interval=%s",
            symbol,
            period,
            interval,
        )
        ticker = self._get_ticker(symbol)
        try:
            history = ticker.history(period=period, interval=interval)
            data = history.to_dict()
            logger.debug(
                "Historical data fetched for %s: %d entries", symbol,␣
↪len(history)
            )
            return data
        except Exception as e:
            logger.error("Error fetching historical data for %s: %s", symbol, e)
            raise

    def fetch_financial_info(self, symbol: str) -> dict:
        """Fetch financial statements for a given ticker symbol."""
        logger.info("Fetching financial info for %s", symbol)
        ticker = self._get_ticker(symbol)
        try:
            financials = {
                "income_statement": ticker.financials,
                "balance_sheet": ticker.balance_sheet,
                "cashflow": ticker.cashflow,
            }
            logger.debug("Financial info fetched for %s", symbol)
            return financials
        except Exception as e:
            logger.error("Error fetching financial info for %s: %s", symbol, e)
            raise

    def fetch_symbol_info(self, symbol: str) -> dict:
        """Fetch general information for a given ticker symbol."""
        logger.info("Fetching symbol info for %s", symbol)
        ticker = self._get_ticker(symbol)
        try:
            info = ticker.info
            data = {
                "name": info.get("longName"),
                "sector": info.get("sector"),
                "industry": info.get("industry"),
                "market_cap": info.get("marketCap"),
                "beta": info.get("beta"),
                "pe_ratio": info.get("trailingPE"),
                "dividend_yield": info.get("dividendYield"),
            }
            logger.debug("Symbol info fetched for %s: %s", symbol, data)
            return data
        except Exception as e:
            logger.error("Error fetching symbol info for %s: %s", symbol, e)
            raise




class EconomicDataTool:
    """Tool for fetching economic indicators from FRED API."""

    name = "EconomicDataTool"
    description = "Fetch economic data (GDP, CPI, unemployment, etc.) via FRED␣
 ↪API."

    # Load configuration once
    api_key: str = fred_api_key
    fred_endpoint: str = fred_api_endpoint

    def fetch_series(
        self, series_id: str, start_date: str = None, end_date: str = None
    ) -> dict:
        """
        Fetch time series data from FRED.

        Args:
            series_id (str): Example 'GDP', 'CPIAUCSL', 'UNRATE'
            start_date (str): 'YYYY-MM-DD'
            end_date (str): 'YYYY-MM-DD'

        Returns:
            dict: Time series data
        """
        params = {"series_id": series_id, "api_key": self.api_key, "file_type":␣
 ↪"json"}
        if start_date:
            params["observation_start"] = start_date
        if end_date:
            params["observation_end"] = end_date

        logger.info(
            "Fetching economic series '%s' from %s", series_id, self.
 ↪fred_endpoint
        )
        try:
            response = requests.get(
                f"{self.fred_endpoint}/series/observations", params=params,␣
↪timeout=10
            )
            response.raise_for_status()
            data = response.json()
            if "observations" not in data:
                logger.error("No observations found for series %s: %s",␣
↪series_id, data)
                raise ValueError(f"Error fetching series {series_id}: {data}")
            logger.info(
                "Fetched %d observations for series %s",
                len(data["observations"]),
                series_id,
            )
            return data["observations"]
        except requests.RequestException as e:
            logger.exception("Request failed for series %s: %s", series_id, e)
            raise
```

[19]:

```
# ---------------------------------------------------------------------
# Global model instance (loaded once and reused across agents)
# ---------------------------------------------------------------------
MODEL = GPT4All(model_name="Phi-3-mini-4k-instruct.Q4_0.gguf", device='cpu')

# ---------------------------------------------------------------------
# Setup module-level logger
# ---------------------------------------------------------------------
logger = logging.getLogger(__name__)
mock = True # To invoke without LLM ( for Debugging)

class Agent:
    """Base class for all AI agents in the multi-agent framework."""

    def __init__(
        self, name: str, role: str, model: str = "gguf-model-falcon-q4_0.gguf"
    ):
        self.name = name
        self.role = role
        self.model = MODEL  # use the preloaded model instance
        logger.info("Initialized agent: %s (%s)", self.name, self.role)
        self.task_counter = {}
    # ---------------------------------------------------------------------
    # Core LLM call
    # ---------------------------------------------------------------------
    def call_llm(self, prompt: str) -> str:
        """Send a prompt to the LLM and return its response."""
        if mock:
            return f"Mock response from {self.name}: {prompt[:50]}..."
        try:
            result = self.model.generate(prompt)
            logger.debug("LLM response for %s: %s...", self.name, result[:80])
            return result
        except Exception as e:
            logger.error("LLM call failed for %s: %s", self.name, e,␣
↪exc_info=True)
            return f"Mock response from {self.name}: {prompt[:50]}..."

    # ------------------------------------------------------------------
    # Process & persist memory
    # ------------------------------------------------------------------
    def process(self, prompt: str, **kwargs) -> str:
        """Process a prompt through the agent and store its memory."""
        logger.info(
            "Agent %s processing context: %s",
            self.name,
            kwargs.get("context", "UnknownContext"),
        )

        result = self.call_llm(prompt)

        memory_dto = AgentMemoryDTO(
            agent_name=kwargs.get("agent_name", self.name),
            context=kwargs.get("context", "UnknownContext"),
            inputs=prompt,
            output=result,
            timestamp=datetime.now().isoformat(),
        )

        # Convert DTO to ORM instance
        memory_entry = AgentMemory(**memory_dto.__dict__)
        save_to_db(SessionAgentMemory, memory_entry)
        logger.debug(
            "Memory saved for agent %s under context '%s'",
            self.name,
            kwargs.get("context", "UnknownContext"),
        )
        return result

    # ------------------------------------------------------------------
    # Default task handler
    # ------------------------------------------------------------------
    def run_task(self, input_data: str) -> str:
        """Default generic task executor."""
        logger.info("Agent %s running task with input: %s", self.name,␣
↪input_data[:80])
        prompt = f"Process this as a {self.role}: {input_data}"
        return self.process(prompt)

    # ---------------------------------------------------------------------
    # Send data between agents
    # ---------------------------------------------------------------------
    def send_to(self, other_agent, input_data: str) -> str:
        """Send processed data from one agent to another."""
        logger.info(
            "%s → %s | Data length: %d", self.name, other_agent.name,␣
↪len(input_data)
        )
        return other_agent.run_task(input_data)
```

```
Downloading: 100%|      | 2.18G/2.18G [00:26<00:00, 80.9MiB/s]
Verifying: 100%|      | 2.18G/2.18G [00:07<00:00, 305MiB/s]
```

[20]:
```
class Coordinator(Agent):
    def __init__(self, model_type: str = "mock"):
        super().__init__("Boss", "coordinator", model_type)
        self.team = []

    def add_agent(self, agent):
        """Add an agent to the coordination team."""
        self.team.append(agent)
        logger.info(f"  Added {agent.name} to team.")

    def delegate_project(self, project_description):
        """Create a research plan and delegate tasks to all team members."""
        logger.info(f"[{self.name}] Delegating project: {project_description}")

        # Generate plan
        plan_prompt = f"Create a step-by-step research plan for:␣
↪{project_description}"
        plan_kwargs = {"agent_name": self.name, "context": "delegate_project"}

        plan = self.process(plan_prompt, **plan_kwargs)
        logger.info(f"[{self.name}] Generated project plan.")

        # Delegate tasks to each team member
        results = {}
        for agent in self.team:
            task = f"Work on project '{project_description}' using your {agent.
↪role} skills."
```

```
            task_kwargs = {"agent_name": agent.name, "context":␣
↪"delegate_project"}

            logger.info(f"[{self.name}] Assigning task to {agent.name} ({agent.
↪role})")
            results[agent.name] = agent.process(task, **task_kwargs)
            logger.info(f"[{agent.name}] Task completed.")

        logger.info(
            f"[{self.name}] Delegation complete for project:␣
↪{project_description}"
        )
        return {"plan": plan, "results": results}




class DecisionMaker(Agent):
    def __init__(self, model_type: str = "mock"):
        super().__init__("DecisionMaker", "investment decision analyst",␣
↪model_type)
        logger.info("Initialized DecisionMaker agent")

    def make_decision(
        self, query: str, senior_summary: str, analyst_summary: str
    ) -> str:
        """
        Analyze the senior researcher's summary and analyst's insights to␣
↪produce
        actionable investment decisions.
        Returns the raw model output as a string.
        """
        logger.info("DecisionMaker analyzing query: %s", query)

        prompt = f"""
        You are an investment decision analyst.
        Based on the following inputs, suggest an action for the query {query}.
        Choose one of Buy, Hold, or Sell.
        Provide a short rationale (1-2 sentences) and risk level (Low, Medium,␣
↪High).

        Senior Researcher's Summary:
        {senior_summary}

        Analyst's Insights:
```

```
        {analyst_summary}
        """

        kwargs = {"agent_name": self.name, "context": "make_decision"}

        decision = self.process(prompt, **kwargs)

        logger.info("DecisionMaker completed decision for query: %s", query)
        logger.debug("DecisionMaker output: %s", decision[:300])

        return decision

    def run_task(self, input_data: tuple) -> str:
        """
        Receive input data (query, senior_summary, analyst_summary)
        and produce an investment decision.
        """
        if not isinstance(input_data, (tuple, list)) or len(input_data) != 3:
            logger.error("Invalid input format for DecisionMaker: %s",␣
 ↪input_data)
            raise ValueError(
                "Expected input_data as a tuple: (query, senior_summary,␣
 ↪analyst_summary)"
            )

        query, senior_summary, analyst_summary = input_data
        logger.debug("Running DecisionMaker task for query: %s", query)
        return self.make_decision(query, senior_summary, analyst_summary)




class Evaluator(Agent):
    def __init__(self, model_type: str = "mock"):
        super().__init__("Evaluator", "evaluation agent", model_type)
        logger.info("Initialized Evaluator agent")

    def evaluate(
        self,
        query: str,
        senior_summary: str,
        analyst_summary: str,
        decision_output: dict,
    ) -> dict:
        logger.info("Evaluator evaluating query: %s", query)
        prompt = f"""
        You are an evaluator for investment research.
        Evaluate the following for query: {query}:

        Senior Researcher's Summary:
        {senior_summary}

        Analyst's Insights:
        {analyst_summary}

        Decision Maker's Suggestion:
        {decision_output}

        Check for:
        - Completeness
        - Clarity
        - Accuracy
        - Suggestions for improvement

        Provide concise feedback.
        """
        kwargs = {"agent_name": self.name, "context": "evaluate"}
        feedback_text = self.process(prompt, **kwargs)
        return {
            "senior_summary_feedback": feedback_text,
            "analyst_summary_feedback": feedback_text,
            "decision_feedback": feedback_text,
        }

    def run_task(self, input_data: tuple) -> dict:
        return self.evaluate(*input_data)


class Optimizer(Agent):
    def __init__(self, model_type: str = "mock"):
        super().__init__("Optimizer", "optimization agent", model_type)
        logger.info("Initialized Optimizer agent")

    def optimize(
        self,
        query: str,
        senior_summary: str,
        analyst_summary: str,
        decision_output: dict,
        evaluation_feedback: dict,
    ) -> str:
        logger.info("Optimizer optimizing query: %s", query)
        prompt = f"""
        You are an optimizer for investment research.
        Improve the summaries based on evaluation feedback:

        Senior Summary: {senior_summary}
        Analyst Insights: {analyst_summary}
        Decision Output: {decision_output}
        Evaluation Feedback: {evaluation_feedback}

        Provide a refined summary or actionable recommendations.
        """
        kwargs = {"agent_name": self.name, "context": "optimize"}
        return self.process(prompt, **kwargs)

    def run_task(self, input_data: tuple) -> str:
        query, senior_summary, analyst_summary, decision_output,␣
 ↪evaluation_feedback = (
            input_data
        )
        return self.optimize(
            query, senior_summary, analyst_summary, decision_output,␣
 ↪evaluation_feedback
        )




class SeniorResearcher(Agent):
    def __init__(self, model_type: str = "mock"):
        super().__init__("SeniorResearcher", "lead researcher", model_type)
        self.stock_agent = StockAgent()
        self.news_agent = NewsAgent()
        self.economic_agent = EconomicAgent()

    def select_research_source(self, query: str) -> list[str]:
        prompt = f"""
        You are a senior researcher. Decide the best research source(s) for␣
 ↪this query: {query}.
        Options: Stock (ticker data), News (recent or historical headlines),␣
 ↪Economic data (GDP, CPI, unemployment, etc.)
        You can choose more than one. Return the selected sources as a␣
 ↪comma-separated list.
        """
        kwargs = {"agent_name": self.name, "context": "select_research_source"}
        choice_text = self.process(prompt, **kwargs)

        sources = [c.strip().lower() for c in choice_text.split(",")]
        valid_sources = {"stock", "news", "economic"}
        selected_sources = [s for s in sources if s in valid_sources]
        if not selected_sources:
            selected_sources = ["news", "economic"]

        logger.info(
            "Selected research sources for query '%s': %s", query,␣
↪selected_sources
        )
        return selected_sources

    def research_stock(self, query: str) -> str:
        logger.info("--- Senior Researcher Delegating Tasks for query: %s ---",␣
↪query)
        selected_sources = self.select_research_source(query)
        stock_summary, news_summary, economic_summary = "", "", ""

        if "stock" in selected_sources:
            stock_summary = self.stock_agent.run_task(query)

        if "news" in selected_sources:
            news_summary = self.news_agent.run_task(query)

        if "economic" in selected_sources:
            economic_prompt = f"Provide economic indicators relevant to {query}␣
↪and choose an appropriate timeframe."
            economic_summary = self.economic_agent.run_task(economic_prompt)

        raw_combined_text = ""
        if stock_summary:
            raw_combined_text += f"Stock Summary:\n{stock_summary}\n\n"
        if news_summary:
            raw_combined_text += f"News Summary:\n{news_summary}\n\n"
        if economic_summary:
            raw_combined_text += f"Economic Summary:\n{economic_summary}"

        prompt = f"""
        You are a senior researcher. Based on the following summaries,
        provide a concise, professional report highlighting key insights,
        trends, risks, and potential opportunities for {query}.
        {raw_combined_text}
        """
        kwargs = {"agent_name": self.name, "context": "research_stock"}
        final_summary = self.process(prompt, **kwargs)

        logger.info("--- Senior Researcher Final Summary for query '%s' ---",␣
↪query)
        return final_summary
    def run_task(self, input_data: str) -> str:
        return self.research_stock(input_data)


# --- Child Agents ---
class StockAgent(Agent):
    def __init__(self):
        super().__init__("StockAgent", "stock data analyst", "mock")
        self.tool = StockTool()

    def fetch_and_summarize(
        self, symbol: str, period: str = "1mo", interval: str = "1d"
    ) -> str:
        logger.info("Fetching stock data for symbol: %s", symbol)
        try:
            history = self.tool.fetch_historical_data(
                symbol, period=period, interval=interval
            )
            history_text = ", ".join(
                f"{d}: {c}"
                for d, c in zip(
                    list(history["Close"].keys())[-10:],
                    list(history["Close"].values())[-10:],
                )
            )
            financials = self.tool.fetch_financial_info(symbol)
            info = self.tool.fetch_symbol_info(symbol)
            financial_summary = f"Market Cap: {info.get('market_cap')}, PE:␣
↪{info.get('pe_ratio')}, Dividend Yield: {info.get('dividend_yield')}"
            prompt = f"""
            You are a stock data analyst. Summarize the stock performance for␣
↪{symbol} based on:
            Recent closes: {history_text}
            Financial metrics: {financial_summary}
            """
            kwargs = {"agent_name": self.name, "context": "fetch_and_summarize"}
            summary_text = self.process(prompt, **kwargs)
            return summary_text
        except Exception as e:
            logger.exception("Error fetching/summarizing data for %s: %s",␣
↪symbol, e)
            return f"Error fetching/summarizing data for {symbol}: {e}"

    def run_task(self, input_data: str) -> str:
        return self.fetch_and_summarize(input_data)
```

```
class NewsAgent(Agent):
    def __init__(self):
        super().__init__("NewsAgent", "financial news analyst", "mock")
        self.tool = NewsTool()

    def select_news_source(self, query: str) -> str:
        prompt = f"""
        You are a financial news analyst. Decide the best news source for this␣
↪query: {query}.
        Options: recent, history, db. Return one word.
        """
        kwargs = {"agent_name": self.name, "context": "select_news_source"}
        choice = self.process(prompt, **kwargs).strip().lower()
        if choice not in ["recent", "history", "db"]:
            choice = "recent"
        logger.info("Selected news source for query '%s': %s", query, choice)
        return choice

    def fetch_and_summarize(self, query: str, **kwargs) -> str:
        source = self.select_news_source(query)
        if "q" not in kwargs:
            kwargs["q"] = query
        articles = self.tool.fetch_news(source=source, **kwargs)
        raw_text = ""
        for a in articles[:10]:
            raw_text += f"Title: {a.title}\nDescription: {a.
↪description}\nPublished: {a.publishedAt}\n\n"

        prompt = f"""
        You are a financial news sentiment analyst.
        Analyze the sentiment of the following news articles for the company/
↪query '{query}'.
        Provide a concise summary of overall sentiment (positive, negative,␣
↪neutral),
        highlight key drivers, and include examples from the articles:
        {raw_text}
        """
        kwargs = {"agent_name": self.name, "context":␣
↪"fetch_and_analyze_sentiment"}
        sentiment_summary = self.process(prompt, **kwargs)
        logger.info("Sentiment analysis complete for query '%s'", query)
        return sentiment_summary

    def run_task(self, input_data: str) -> str:
        return self.fetch_and_summarize(input_data)
```

```python
class EconomicAgent(Agent):
    def __init__(self):
        super().__init__("EconomicAgent", "economic data analyst", "mock")
        self.tool = EconomicDataTool()

    def fetch_and_summarize(
        self, series_ids: list, start_date: str = None, end_date: str = None
    ) -> str:
        raw_text = ""
        for series in series_ids:
            try:
                observations = self.tool.fetch_series(series, start_date,␣
 ↪end_date)
                if observations:
                    obs_text = ", ".join(
                        f"{obs['date']}:{obs['value']}" for obs in␣
 ↪observations[-10:]
                    )
                    raw_text += f"{series}: {obs_text}\n"
            except Exception as e:
                logger.exception("Error fetching economic data for %s: %s",␣
 ↪series, e)
                raw_text += f"{series}: Error fetching data\n"

        prompt = f"""
        You are an economic data analyst. Summarize this economic data in a␣
 ↪concise, professional format:
        {raw_text}
        """
        summary_text = self.process(prompt)
        logger.info("Economic summarization complete for series: %s",␣
 ↪series_ids)
        return summary_text

    def run_task(self, input_data) -> str:
        series_ids = input_data if isinstance(input_data, list) else␣
 ↪[input_data]
        return self.fetch_and_summarize(series_ids)



class Analyst(Agent):
    def __init__(self, model_type: str = "mock"):
        super().__init__("Analyst", "financial analyst", model_type)

    def run_task(self, research_summary: str) -> str:
        """
        Receive input from Researcher and provide analysis insights.
        """
        prompt = f"""
        You are a financial analyst. Based on the following research summaries,
        provide insights on trends, risks, and potential opportunities:

        {research_summary}
        """
        kwargs = {"agent_name": self.name, "context": "run_task"}

        logger.info(f"[{self.name}] Starting analysis with context: {kwargs}")
        analysis = self.process(prompt, **kwargs)
        logger.info(f"[{self.name}] Analysis complete. Output: {analysis[:200]}.
↪..")

        return analysis

class Writer(Agent):
    def __init__(self, model_type: str = "mock"):
        super().__init__("Writer", "content writer", model_type)
        logger.info("Initialized Writer agent")

    def write_report(self, content: str) -> str:
        """
        Generate a professional report based on the given content.
        """
        logger.info("Writer received content to generate report")
        kwargs = {"agent_name": self.name, "context": "write_report"}

        report = self.process(
            f"Write a professional report based on this analysis:\n{content}",␣
↪**kwargs
        )

        logger.info("Writer completed report generation")
        logger.debug("Report content (truncated 300 chars): %s", report[:300])
        return report

    def run_task(self, input_data: str) -> str:
        logger.info("Writer running task")
        return self.write_report(input_data)
```

[21]:
```
class MultiAgentTeam:
    def __init__(self):
        # Instantiate agents
        self.coordinator = Coordinator()
        self.researcher = SeniorResearcher()
        self.analyst = Analyst()
        self.decision_maker = DecisionMaker()
        self.evaluator = Evaluator()
        self.optimizer = Optimizer()
        self.writer = Writer()

        # Wire the team
        self.coordinator.add_agent(self.researcher)
        self.coordinator.add_agent(self.analyst)
        self.coordinator.add_agent(self.decision_maker)
        self.coordinator.add_agent(self.evaluator)
        self.coordinator.add_agent(self.optimizer)
        self.coordinator.add_agent(self.writer)


        logger.info("Multi-agent team created!")

    def execute_project(self, project_description: str):
        logger.info(f"EXECUTING PROJECT: {project_description}")
        logger.info("=" * 60)

        result = {}
        try:
            # Step 1: Coordinator delegates project
            result["coordination"] = self.coordinator.delegate_project(
                project_description
            )
            print(result["coordination"])
        except Exception as e:
            result["coordination"] = f"[Error] {e}"
            logger.error("Coordinator error: %s", e)

        try:
            # Step 2: SeniorResearcher gathers research
            result["research"] = self.researcher.
↪research_stock(project_description)
        except Exception as e:
            result["research"] = f"[Error] {e}"
            logger.error("Researcher error: %s", e)

        try:
            # Step 3: Analyst processes the research
            analyst_inputs = result.get("research", "")
            result["analysis"] = self.researcher.send_to(self.analyst,␣
↪analyst_inputs)
        except Exception as e:
            result["analysis"] = f"[Error] {e}"
            logger.error("Analyst error: %s", e)

        try:
            # Step 4: DecisionMaker generates decision/recommendation
            inputs = (
                project_description,
                result.get("research", ""),
                result.get("analysis", ""),
            )
            result["decision"] = self.analyst.send_to(self.decision_maker,␣
↪inputs)
        except Exception as e:
            result["decision"] = f"[Error] {e}"
            logger.error("DecisionMaker error: %s", e)

        try:
            # Step 5a: Evaluator reviews the outputs
            eval_inputs = (*inputs, result.get("decision", ""))
            evaluation_feedback = self.decision_maker.send_to(
                self.evaluator, eval_inputs
            )

            # Step 5b: Optimizer refines summaries based on evaluation
            opt_inputs = (*inputs, result.get("decision", ""),␣
↪evaluation_feedback)
            result["raw_output"] = self.evaluator.send_to(
                self.optimizer, opt_inputs
            )

        except Exception as e:
            result["raw_output"] = f"[Error] {e}"
            logger.error("Evaluator or Optimizer error: %s", e)

        try:
            # Step 6: Writer agent
            result["report"] = self.optimizer.send_to(
                self.writer, result["raw_output"]
            )
        except Exception as e:
            result["report"] = f"[Error] {e}"
            logger.error("Writer error: %s", e)
        logger.info("PROJECT EXECUTION COMPLETED!")
        return result

    def show_team_status(self):
        logger.info("TEAM STATUS")
        agents = [
        self.coordinator,
        self.researcher,
        self.analyst,
        self.decision_maker,
        self.evaluator,
        self.optimizer,
        self.writer
    ]

        for agent in agents:
            tasks_completed = len(getattr(agent, "memory", {}))
            logger.info(
                "  %s (%s): %d tasks completed",
                agent.name,
                getattr(agent, "role", "N/A"),
                tasks_completed,
            )
```

[22]:
```
if __name__ == "__main__":
    # Initialize DB
    init_db()

    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
    )
    team = MultiAgentTeam()
    project_result = team.execute_project("Analyze AAPL")
    team.show_team_status()

    logger.info("Project Result Summary:")
    for k, v in project_result.items():
        logger.info("%s: %s", k, v)
```

```
ERROR:__main__:Request failed for endpoint https://newsapi.org/v2/top-headlines:
401 Client Error: Unauthorized for url: https://newsapi.org/v2/top-
headlines?q=Analyze+AAPL&apiKey=NEWS_API_KEY
```

```
{'plan': 'Mock response from Boss: Create a step-by-step research plan for:
Analyze A…', 'results': {'SeniorResearcher': "Mock response from
SeniorResearcher: Work on project 'Analyze AAPL' using your lead res…",
'Analyst': "Mock response from Analyst: Work on project 'Analyze AAPL' using
your financia…", 'DecisionMaker': "Mock response from DecisionMaker: Work on
project 'Analyze AAPL' using your investme…", 'Evaluator': "Mock response from
Evaluator: Work on project 'Analyze AAPL' using your evaluati…", 'Optimizer':
"Mock response from Optimizer: Work on project 'Analyze AAPL' using your
optimiza…", 'Writer': "Mock response from Writer: Work on project 'Analyze
AAPL' using your content …"}}

ERROR:__main__:Request failed for series Provide economic indicators relevant to
Analyze AAPL and choose an appropriate timeframe.: 400 Client Error: Bad Request
for url: https://api.stlouisfed.org/fred/series/observations?series_id=Provide+e
conomic+indicators+relevant+to+Analyze+AAPL+and+choose+an+appropriate+timeframe.
&api_key=FRES_API_KEY&file_type=json
Traceback (most recent call last):
  File "/tmp/ipython-input-3110710531.py", line 189, in fetch_series
    response.raise_for_status()
  File "/usr/local/lib/python3.12/dist-packages/requests/models.py", line 1026,
in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 400 Client Error: Bad Request for url: https://ap
i.stlouisfed.org/fred/series/observations?series_id=Provide+economic+indicators+
relevant+to+Analyze+AAPL+and+choose+an+appropriate+timeframe.&api_key=FRES_API_K
EY&file_type=json
ERROR:__main__:Error fetching economic data for Provide economic indicators
relevant to Analyze AAPL and choose an appropriate timeframe.: 400 Client Error:
Bad Request for url: https://api.stlouisfed.org/fred/series/observations?series_
id=Provide+economic+indicators+relevant+to+Analyze+AAPL+and+choose+an+appropriat
e+timeframe.&api_key=FRES_API_KEY&file_type=json
Traceback (most recent call last):
  File "/tmp/ipython-input-1644020895.py", line 340, in fetch_and_summarize
    observations = self.tool.fetch_series(series, start_date, end_date)
                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "/tmp/ipython-input-3110710531.py", line 189, in fetch_series
    response.raise_for_status()
  File "/usr/local/lib/python3.12/dist-packages/requests/models.py", line 1026,
in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 400 Client Error: Bad Request for url: https://ap
i.stlouisfed.org/fred/series/observations?series_id=Provide+economic+indicators+
relevant+to+Analyze+AAPL+and+choose+an+appropriate+timeframe.&api_key=FRES_API_K
EY&file_type=json
```

[23]:
```
# ============================================================
# WORKFLOW OVERVIEW
# ============================================================

# Coordinator → SeniorResearcher → (StockAgent / NewsAgent / EconomicAgent)
# → Analyst → DecisionMaker → Evaluator → Optimizer → Writer

# Flow Explanation:
# 1. Coordinator: Plans research steps for a given stock symbol.
# 2. SeniorResearcher: Routes the task to the right specialists.
# 3. StockAgent / NewsAgent / EconomicAgent: Fetch and process data.
# 4. Analyst: Analyzes research summaries and identifies insights.
# 5. DecisionMaker: Generates actionable investment decisions.
# 6. Evaluator: Evaluates the quality and coherence of decisions.
# 7. Optimizer: Refines decisions using evaluator feedback.
# 8. Writer: Compiles the final investment report.
```

[24]:
```
# Agent Functions:
# 1. Plans its research steps for a given stock symbol:
#    - The `Coordinator` agent's `delegate_project` method generates a research␣
↪plan. (See `Coordinator` class in the code.)
# 2. Uses tools dynamically (APIs, datasets, retrieval):
#    - The `StockAgent` uses `StockTool` (yfinance).
#    - The `NewsAgent` uses `NewsTool` (NewsAPI and local DB).
#    - The `EconomicAgent` uses `EconomicDataTool` (FRED API). (See␣
↪`StockAgent`, `NewsAgent`, and `EconomicAgent` classes.)
# 3. Self-reflects to assess the quality of its output:
#    - The `Evaluator` agent's `evaluate` method assesses the quality of the␣
↪research output. (See `Evaluator` class.)
# 4. Learns across runs (e.g., keeps brief memories or notes to improve future␣
↪analyses):
#    - The `Agent` base class includes a `process` method that saves inputs,␣
↪outputs, and context to the `AgentMemory` database. (See `Agent` class and␣
↪the database setup.)

# Workflow Patterns:
# 1. Prompt Chaining: Ingest News -> Preprocess -> Classify -> Extract ->␣
↪Summarize
#    - This pattern is demonstrated in the `NewsAgent`'s `fetch_and_summarize`␣
↪method, which fetches articles, processes the raw text, and then uses a␣
↪prompt to analyze sentiment and summarize. (See `NewsAgent` class.)
# 2. Routing: Direct content to the right specialist (e.g., earnings, news, or␣
↪market analyzers)
#    - The `SeniorResearcher` agent's `select_research_source` method␣
↪determines which child agents (`StockAgent`, `NewsAgent`, `EconomicAgent`)␣
↪are relevant to the query and routes the task accordingly. (See␣
↪`SeniorResearcher` class.)
#    - The `MultiAgentTeam` orchestrates the flow between the␣
↪`SeniorResearcher`, `Analyst`, `DecisionMaker`, `Evaluator`, `Optimizer`,␣
↪and `Writer` agents. (See `MultiAgentTeam` class.)
# 3. Evaluator-Optimizer: Generate analysis -> evaluate quality -> refine using␣
↪feedback.
#    - This pattern is implemented through the interaction of the␣
↪`DecisionMaker` (generates decision), `Evaluator` (evaluates the decision␣
↪and summaries), and `Optimizer` (refines output based on feedback). (See␣
↪`MultiAgentTeam`'s `execute_project` method and the `Evaluator` and␣
↪`Optimizer` classes.)
```

[25]:
```
# END
```