```
import asyncio
import os
from concurrent.futures import ThreadPoolExecutor, as_completed
from typing import Any, Dict, List, Union

import aiohttp
import requests
from dotenv import load_dotenv
from loguru import logger
from swarm_models import OpenAIChat

from swarms import Agent

load_dotenv()

# New Pharmaceutical Agent System Prompt
PHARMA_AGENT_SYS_PROMPT = """
You are a pharmaceutical data analysis agent specializing in retrieving and analyzing chemical data.

You have access to the latest chemical databases and can provide detailed analysis of any chemical compounds
relevant to pharmaceutical research. Your goal is to assist pharmaceutical companies in retrieving chemical
properties, safety data, and usage details for various compounds.

When given a chemical name, you will:
```

1. Retrieve the relevant chemical properties such as molecular weight, CAS number, chemical formula,
melting point, boiling point, and solubility.
2. Analyze the chemical properties and provide insights on the compound's potential applications in pharmaceuticals, safety precautions, and any known interactions with other compounds.
3. If you encounter missing or incomplete data, make a note of it and proceed with the available information,
ensuring you provide the most relevant and accurate analysis.

You will respond in a structured format and, where applicable, recommend further reading or research papers.
Keep responses concise but informative, with a focus on helping pharmaceutical companies make informed decisions
about chemical compounds.

If there are specific safety risks or regulatory concerns, highlight them clearly.
"""

class PharmaAgent:
"""
A pharmaceutical data agent that dynamically fetches chemical data from external sources and uses an LLM
to analyze and respond to queries related to chemicals for pharmaceutical companies.

Attributes:

```
    api_key (str): The OpenAI API key for accessing the LLM.
    agent (Agent): An instance of the swarms Agent class to manage interactions with the LLM.
"""

def __init__(
    self,
    model_name: str = "gpt-4o-mini",
    temperature: float = 0.1,
):
    """
    Initializes the PharmaAgent with the OpenAI model and necessary configurations.

    Args:
        model_name (str): The name of the LLM model to use.
        temperature (float): The temperature for the LLM to control randomness.
    """
    self.api_key = os.getenv("OPENAI_API_KEY")
    logger.info("Initializing OpenAI model and Agent...")
    model = OpenAIChat(
        openai_api_key=self.api_key,
        model_name=model_name,
        temperature=temperature,
    )

    # Initialize the agent
    self.agent = Agent(
```

```
        agent_name="Pharmaceutical-Data-Agent",
        system_prompt=PHARMA_AGENT_SYS_PROMPT,
        llm=model,
        max_loops=1,
        autosave=True,
        dashboard=False,
        verbose=True,
        dynamic_temperature_enabled=True,
        saved_state_path="pharma_agent.json",
        user_name="swarms_corp",
        retry_attempts=1,
        context_length=200000,
        return_step_meta=False,
    )
    logger.info("Agent initialized successfully.")


def get_latest_chemical_data(
    self, chemical_name: str
) -> Union[Dict[str, Any], Dict[str, str]]:
    """
    Fetches the latest chemical data dynamically from PubChem's API.

    Args:
        chemical_name (str): The name of the chemical to query.

    Returns:
```

```
        Dict[str, Any]: A dictionary containing chemical data if successful, or an error message if failed.
    """
    logger.info(f"Fetching data for chemical: {chemical_name}")
    base_url = (
        "https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/name"
    )
    response = requests.get(f"{base_url}/{chemical_name}/JSON")

    if response.status_code == 200:
        chemical_data = response.json()
        try:
            compound_info = chemical_data["PC_Compounds"][0]
            chemical_properties = {
                "name": compound_info.get("props", [])[0]
                .get("urn", {})
                .get("label", "Unknown"),
                "molecular_weight": compound_info.get(
                    "props", []
                )[1]
                .get("value", {})
                .get("fval", "Unknown"),
                "CAS_number": compound_info.get("props", [])[2]
                .get("urn", {})
                .get("label", "Unknown"),
                "formula": compound_info.get("props", [])[3]
```

```
            .get("value", {})
            .get("sval", "Unknown"),
            "properties": {
                "melting_point": compound_info.get(
                    "props", []
                )[4]
                .get("value", {})
                .get("fval", "Unknown"),
                "boiling_point": compound_info.get(
                    "props", []
                )[5]
                .get("value", {})
                .get("fval", "Unknown"),
                "solubility": "miscible with water",  # Placeholder as PubChem may not provide this
            },
        }
        logger.info(
            f"Data successfully retrieved for chemical: {chemical_name}"
        )
        return chemical_properties
    except (IndexError, KeyError):
        logger.error(
            f"Incomplete data for chemical: {chemical_name}"
        )
        return {
            "error": "Chemical data not found or incomplete"
```

```
            }
        else:
            logger.error(
                f"Failed to fetch chemical data. Status code: {response.status_code}"
            )
            return {
                "error": f"Failed to fetch chemical data. Status code: {response.status_code}"
            }

    def query_chemical_data(self, chemical_name: str) -> str:
        """
        Queries the latest chemical data and passes it to the LLM agent for further analysis and response.

        Args:
            chemical_name (str): The name of the chemical to query.

        Returns:
            str: The response from the LLM agent after analyzing the chemical data.
        """
        chemical_data = self.get_latest_chemical_data(chemical_name)
        if "error" in chemical_data:
            return f"Error: {chemical_data['error']}"

        prompt = f"Fetch and analyze the latest chemical data for {chemical_name}: {chemical_data}"
        logger.info(
```

```
        f"Sending chemical data to agent for analysis: {chemical_name}"
    )
    return self.agent.run(prompt)


def run(self, chemical_name: str) -> str:
    """
    Main method to fetch and analyze the latest chemical data using the LLM agent.

    Args:
        chemical_name (str): The name of the chemical to query.

    Returns:
        str: The result of the chemical query processed by the agent.
    """
    logger.info(f"Running chemical query for: {chemical_name}")
    return self.query_chemical_data(chemical_name)


def run_concurrently(
    self, chemical_names: List[str]
) -> List[str]:
    """
    Runs multiple chemical queries concurrently using ThreadPoolExecutor.

    Args:
        chemical_names (List[str]): List of chemical names to query.
```

```
    Returns:
        List[str]: List of results from the LLM agent for each chemical.
    """
    logger.info("Running chemical queries concurrently...")
    results = []
    with ThreadPoolExecutor() as executor:
        future_to_chemical = {
            executor.submit(self.run, chemical): chemical
            for chemical in chemical_names
        }
        for future in as_completed(future_to_chemical):
            chemical = future_to_chemical[future]
            try:
                result = future.result()
                logger.info(f"Completed query for: {chemical}")
                results.append(result)
            except Exception as exc:
                logger.error(
                    f"Chemical {chemical} generated an exception: {exc}"
                )
                results.append(f"Error querying {chemical}")
    return results


async def fetch_chemical_data_async(
    self, session: aiohttp.ClientSession, chemical_name: str
) -> Union[Dict[str, Any], Dict[str, str]]:
```

```
    """
    Asynchronously fetches chemical data using aiohttp.

    Args:
        session (aiohttp.ClientSession): An aiohttp client session.
        chemical_name (str): The name of the chemical to query.

    Returns:
        Union[Dict[str, Any], Dict[str, str]]: A dictionary containing chemical data if successful, or an error message if failed.
    """
    logger.info(
        f"Fetching data asynchronously for chemical: {chemical_name}"
    )
    base_url = (
        "https://pubchem.ncbi.nlm.nih.gov/rest/pug/compound/name"
    )
    async with session.get(
        f"{base_url}/{chemical_name}/JSON"
    ) as response:
        if response.status == 200:
            chemical_data = await response.json()
            try:
                compound_info = chemical_data["PC_Compounds"][0]
                chemical_properties = {
                    "name": compound_info.get("props", [])[0]
```

```
.get("urn", {})
.get("label", "Unknown"),
"molecular_weight": compound_info.get(
    "props", []
)[1]
.get("value", {})
.get("fval", "Unknown"),
"CAS_number": compound_info.get("props", [])[
    2
]
.get("urn", {})
.get("label", "Unknown"),
"formula": compound_info.get("props", [])[3]
.get("value", {})
.get("sval", "Unknown"),
"properties": {
    "melting_point": compound_info.get(
        "props", []
    )[4]
    .get("value", {})
    .get("fval", "Unknown"),
    "boiling_point": compound_info.get(
        "props", []
    )[5]
    .get("value", {})
    .get("fval", "Unknown"),
```

```
                    "solubility": "miscible with water",  # Placeholder as PubChem may not provide this
                },
            }
            logger.info(
                f"Data successfully retrieved for chemical: {chemical_name}"
            )
            return chemical_properties
        except (IndexError, KeyError):
            logger.error(
                f"Incomplete data for chemical: {chemical_name}"
            )
            return {
                "error": "Chemical data not found or incomplete"
            }
    else:
        logger.error(
            f"Failed to fetch chemical data. Status code: {response.status}"
        )
        return {
            "error": f"Failed to fetch chemical data. Status code: {response.status}"
        }


async def run_async(self, chemical_name: str) -> str:
    """
    Asynchronously runs the agent to fetch and analyze the latest chemical data.
```

```
        Args:
            chemical_name (str): The name of the chemical to query.

        Returns:
            str: The result of the chemical query processed by the agent.
        """
        async with aiohttp.ClientSession() as session:
            chemical_data = await self.fetch_chemical_data_async(
                session, chemical_name
            )
            if "error" in chemical_data:
                return f"Error: {chemical_data['error']}"

                    prompt = f"Fetch and analyze the latest chemical data for {chemical_name}:
{chemical_data}"
            logger.info(
                f"Sending chemical data to agent for analysis: {chemical_name}"
            )
            return self.agent.run(prompt)

    async def run_many_async(
        self, chemical_names: List[str]
    ) -> List[str]:
        """
        Runs multiple chemical queries asynchronously using aiohttp and asyncio.
```

```
        Args:
            chemical_names (List[str]): List of chemical names to query.

        Returns:
            List[str]: List of results from the LLM agent for each chemical.
        """
        logger.info(
            "Running multiple chemical queries asynchronously..."
        )
        tasks = []
        async with aiohttp.ClientSession():
            for chemical in chemical_names:
                task = self.run_async(chemical)
                tasks.append(task)
            return await asyncio.gather(*tasks)


# Example usage
if __name__ == "__main__":
    pharma_agent = PharmaAgent()

    # Example of running concurrently
    chemical_names = ["formaldehyde", "acetone", "ethanol"]
    concurrent_results = pharma_agent.run_concurrently(chemical_names)
    print(concurrent_results)
```