```
import asyncio
import os
import uuid
from datetime import datetime
from typing import Any, Dict, List, Optional

import chromadb
from dotenv import load_dotenv
from loguru import logger
from pydantic import BaseModel, Field
from swarm_models import OpenAIChat

from swarms import Agent
from swarms.prompts.finance_agent_sys_prompt import (
    FINANCIAL_AGENT_SYS_PROMPT,
)

load_dotenv()

# Initialize ChromaDB client
chroma_client = chromadb.Client()

# Create a ChromaDB collection to store tasks, responses, and all swarm activity
swarm_collection = chroma_client.create_collection(
    name="swarm_activity"
)
```

```
class InteractionLog(BaseModel):
    """
    Pydantic model to log all interactions between agents, tasks, and responses.
    """

    interaction_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()),
        description="Unique ID for the interaction.",
    )
    agent_name: str
    task: str
    timestamp: datetime = Field(default_factory=datetime.utcnow)
    response: Optional[Dict[str, Any]] = None
    status: str = Field(
        description="The status of the interaction, e.g., 'completed', 'failed'."
    )
    neighbors: Optional[List[str]] = (
        None  # Names of neighboring agents involved
    )
    conversation_id: Optional[str] = Field(
        default_factory=lambda: str(uuid.uuid4()),
        description="Unique ID for the conversation history.",
    )
```

```
class AgentHealthStatus(BaseModel):
    """
    Pydantic model to log and monitor agent health.
    """

    agent_name: str
    timestamp: datetime = Field(default_factory=datetime.utcnow)
    status: str = Field(
        default="available",
        description="Agent health status, e.g., 'available', 'busy', 'failed'.",
    )
    active_tasks: int = Field(
        0,
        description="Number of active tasks assigned to this agent.",
    )
    load: float = Field(
        0.0,
        description="Current load on the agent (CPU or memory usage).",
    )


class Swarm:
    """
      A scalable swarm architecture where agents can communicate by posting and querying all activities to ChromaDB.
```

Every input task, response, and action by the agents is logged to the vector database for persistent tracking.

Attributes:

agents (List[Agent]): A list of initialized agents.

chroma_client (chroma.Client): An instance of the ChromaDB client for agent-to-agent communication.

api_key (str): The OpenAI API key.

health_statuses (Dict[str, AgentHealthStatus]): A dictionary to monitor agent health statuses.

"""

```
def __init__(
    self,
    agents: List[Agent],
    chroma_client: chromadb.Client,
    api_key: str,
) -> None:
    """
```

Initializes the swarm with agents and a ChromaDB client for vector storage and communication.

Args:

agents (List[Agent]): A list of initialized agents.

chroma_client (chroma.Client): The ChromaDB client for handling vector embeddings.

api_key (str): The OpenAI API key.

"""

```
    self.agents = agents
    self.chroma_client = chroma_client
    self.api_key = api_key
    self.health_statuses: Dict[str, AgentHealthStatus] = {
        agent.agent_name: AgentHealthStatus(
            agent_name=agent.agent_name
        )
        for agent in agents
    }
    logger.info(f"Swarm initialized with {len(agents)} agents.")


def _log_to_db(
    self, data: Dict[str, Any], description: str
) -> None:
    """
    Logs a dictionary of data into the ChromaDB collection as a new entry.

    Args:
        data (Dict[str, Any]): The data to log in the database (task, response, etc.).
        description (str): Description of the action (e.g., 'task', 'response').
    """
    logger.info(f"Logging {description} to the database: {data}")
    swarm_collection.add(
        documents=[str(data)],
        ids=[str(uuid.uuid4())],  # Unique ID for each entry
        metadatas=[
```

```
            {
                "description": description,
                "timestamp": datetime.utcnow().isoformat(),
            }
        ],
    )
    logger.info(
        f"{description.capitalize()} logged successfully."
    )

async def _find_most_relevant_agent(
    self, task: str
) -> Optional[Agent]:
    """
        Finds the agent whose system prompt is most relevant to the given task by querying ChromaDB.
    If no relevant agents are found, return None and log a message.

    Args:
        task (str): The task for which to find the most relevant agent.

    Returns:
        Optional[Agent]: The most relevant agent for the task, or None if no relevant agent is found.
    """
    logger.info(
        f"Searching for the most relevant agent for the task: {task}"
```

```
)

# Query ChromaDB collection for nearest neighbor to the task
result = swarm_collection.query(
    query_texts=[task], n_results=4
)

# Check if the query result contains any data
if not result["ids"] or not result["ids"][0]:
    logger.error(
        "No relevant agents found for the given task."
    )
    return None  # No agent found, return None

# Extract the agent ID from the result and find the corresponding agent
agent_id = result["ids"][0][0]
most_relevant_agent = next(
    (
        agent
        for agent in self.agents
        if agent.agent_name == agent_id
    ),
    None,
)

if most_relevant_agent:
```

```
            logger.info(
                f"Most relevant agent for task '{task}' is {most_relevant_agent.agent_name}."
            )
        else:
            logger.error("No matching agent found in the agent list.")

        return most_relevant_agent

    def _monitor_health(self, agent: Agent) -> None:
        """
        Monitors the health status of agents and logs it to the database.

        Args:
            agent (Agent): The agent whose health is being monitored.
        """
        current_status = self.health_statuses[agent.agent_name]
        current_status.active_tasks += (
            1  # Example increment for active tasks
        )
        current_status.status = (
            "busy" if current_status.active_tasks > 0 else "available"
        )
        current_status.load = 0.5  # Placeholder for real load data
        logger.info(
                        f"Agent {agent.agent_name} is currently {current_status.status} with load
{current_status.load}."
```

```
    )

    # Log health status to the database
    self._log_to_db(current_status.dict(), "health status")

def post_message(self, agent: Agent, message: str) -> None:
    """
    Posts a message from an agent to the shared database.

    Args:
        agent (Agent): The agent posting the message.
        message (str): The message to be posted.
    """
    logger.info(
        f"Agent {agent.agent_name} posting message: {message}"
    )
    message_data = {
        "agent_name": agent.agent_name,
        "message": message,
        "timestamp": datetime.utcnow().isoformat(),
    }
    self._log_to_db(message_data, "message")

def query_messages(
    self, query: str, n_results: int = 5
) -> List[Dict[str, Any]]:
```

```
    """
    Queries the database for relevant messages.

    Args:
        query (str): The query message or task for which to retrieve related messages.
        n_results (int, optional): The number of relevant messages to retrieve. Defaults to 5.

    Returns:
        List[Dict[str, Any]]: A list of relevant messages and their metadata.
    """
    logger.info(f"Querying the database for query: {query}")
    results = swarm_collection.query(
        query_texts=[query], n_results=n_results
    )
    logger.info(
        f"Found {len(results['documents'])} relevant messages."
    )
    return results

async def run_async(self, task: str) -> None:
    """
    Main entry point to find the most relevant agent, submit the task, and allow agents to
    query the database to understand the task's history. Logs every task and response.

    Args:
        task (str): The task to be completed.
```

```
    """
    # Query past messages to understand task history
    past_messages = self.query_messages(task)
    logger.info(
        f"Past messages related to task '{task}': {past_messages}"
    )

    # Find the most relevant agent
    agent = await self._find_most_relevant_agent(task)

    if agent is None:
        logger.error(
            f"No relevant agent found for task: {task}. Task submission aborted."
        )
        return  # Exit the function if no relevant agent is found

    # Submit the task to the agent if found
    await self._submit_task_to_agent(agent, task)

async def _submit_task_to_agent(
    self, agent: Agent, task: str
) -> Dict[str, Any]:
    """
    Submits a task to the specified agent and logs the result asynchronously.

    Args:
```

```
    agent (Agent): The agent to which the task will be submitted.
    task (str): The task to be solved.

Returns:
    Dict[str, Any]: The result of the task from the agent.
"""
if agent is None:
    logger.error("No agent provided for task submission.")
    return

logger.info(
    f"Submitting task '{task}' to agent {agent.agent_name}."
)

interaction_log = InteractionLog(
    agent_name=agent.agent_name, task=task, status="started"
)

# Log the task as a message to the shared database
self._log_to_db(
    {"task": task, "agent_name": agent.agent_name}, "task"
)

result = await agent.run(task)

interaction_log.response = result
```

```
        interaction_log.status = "completed"
        interaction_log.timestamp = datetime.utcnow()

        logger.info(
            f"Task completed by agent {agent.agent_name}. Logged interaction: {interaction_log.dict()}"
        )

        # Log the result as a message to the shared database
        self._log_to_db(
            {"response": result, "agent_name": agent.agent_name},
            "response",
        )

        return result

    def run(self, task: str, *args, **kwargs):
        return asyncio.run(self.run_async(task))


# Initialize the OpenAI model and agents
api_key = os.getenv("OPENAI_API_KEY")

model = OpenAIChat(
    openai_api_key=api_key, model_name="gpt-4o-mini", temperature=0.1
)
```

```
# Example agent creation
agent = Agent(
    agent_name="Financial-Analysis-Agent",
    system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
    llm=model,
    max_loops=1,
    autosave=True,
    dashboard=False,
    verbose=True,
    dynamic_temperature_enabled=True,
    saved_state_path="finance_agent.json",
    user_name="swarms_corp",
    retry_attempts=1,
    context_length=200000,
    return_step_meta=False,
)

# Example agents list
agents_list = [agent]

# Create the swarm
swarm = Swarm(
    agents=agents_list, chroma_client=chroma_client, api_key=api_key
)

# Execute tasks asynchronously
```

```
task = "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the criteria?"

print(swarm.run(task))
```