```
from typing import List, Optional


from tenacity import retry, stop_after_attempt, wait_exponential
from typing import Union, Callable, Any
from swarms import Agent
from swarms.utils.loguru_logger import initialize_logger
from swarms.utils.lazy_loader import lazy_import_decorator
from swarms.utils.auto_download_check_packages import (
    auto_check_and_download_package,
)



logger = initialize_logger(log_folder="agent_router")



@lazy_import_decorator
class AgentRouter:
    """
    Initialize the AgentRouter.

    Args:
        collection_name (str): Name of the collection in the vector database.
        persist_directory (str): Directory to persist the vector database.
        n_agents (int): Number of agents to return in queries.
        *args: Additional arguments to pass to the chromadb Client.
        **kwargs: Additional keyword arguments to pass to the chromadb Client.
```

```
"""

def __init__(
    self,
    collection_name: str = "agents",
    persist_directory: str = "./vector_db",
    n_agents: int = 1,
    *args,
    **kwargs,
):
    try:
        import chromadb
    except ImportError:
        auto_check_and_download_package(
            "chromadb", package_manager="pip", upgrade=True
        )
        import chromadb

    self.collection_name = collection_name
    self.n_agents = n_agents
    self.persist_directory = persist_directory
    self.client = chromadb.Client(*args, **kwargs)
    self.collection = self.client.create_collection(
        collection_name
    )
    self.agents: List[Agent] = []
```

```
@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10),
)
def add_agent(self, agent: Agent) -> None:
    """
    Add an agent to the vector database.

    Args:
        agent (Agent): The agent to add.

    Raises:
        Exception: If there's an error adding the agent to the vector database.
    """
    try:
        agent_text = f"{agent.name} {agent.description} {agent.system_prompt}"
        self.collection.add(
            documents=[agent_text],
            metadatas=[{"name": agent.name}],
            ids=[agent.name],
        )
        self.agents.append(agent)
        logger.info(
            f"Added agent {agent.name} to the vector database."
        )
    except Exception as e:
        logger.error(
            f"Error adding agent {agent.name} to the vector database: {str(e)}"
        )
        raise

def add_agents(
    self, agents: List[Union[Agent, Callable, Any]]
) -> None:
    """
    Add multiple agents to the vector database.

    Args:
        agents (List[Union[Agent, Callable, Any]]): List of agents to add.
    """
    for agent in agents:
        self.add_agent(agent)

def update_agent_history(self, agent_name: str) -> None:
    """
    Update the agent's entry in the vector database with its interaction history.

    Args:
        agent_name (str): The name of the agent to update.
    """
    agent = next(
        (a for a in self.agents if a.name == agent_name), None
    )
    if agent:
        history = agent.short_memory.return_history_as_string()
        history_text = " ".join(history)
        updated_text = f"{agent.name} {agent.description} {agent.system_prompt} {history_text}"

        self.collection.update(
            ids=[agent_name],
            documents=[updated_text],
            metadatas=[{"name": agent_name}],
        )
        logger.info(
            f"Updated agent {agent_name} with interaction history."
        )
    else:
        logger.warning(
            f"Agent {agent_name} not found in the database."
        )

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10),
)
def find_best_agent(
    self, task: str, *args, **kwargs
```

```
) -> Optional[Agent]:
    """
    Find the best agent for a given task.

    Args:
        task (str): The task description.
        *args: Additional arguments to pass to the collection.query method.
        **kwargs: Additional keyword arguments to pass to the collection.query method.

    Returns:
        Optional[Agent]: The best matching agent, if found.

    Raises:
        Exception: If there's an error finding the best agent.
    """
    try:
        results = self.collection.query(
            query_texts=[task],
            n_results=self.n_agents,
            *args,
            **kwargs,
        )

        if results["ids"]:
            best_match_name = results["ids"][0][0]
            best_agent = next(
            (
                a
                for a in self.agents
                if a.name == best_match_name
            ),
            None,
        )
        if best_agent:
            logger.info(
                f"Found best matching agent: {best_match_name}"
            )
            return best_agent
        else:
            logger.warning(
                f"Agent {best_match_name} found in index but not in agents list."
            )
    else:
        logger.warning(
            "No matching agent found for the given task."
        )

    return None
except Exception as e:
    logger.error(f"Error finding best agent: {str(e)}")
    raise
```

```
# # Example usage
# if __name__ == "__main__":
#     from dotenv import load_dotenv
#     from swarm_models import OpenAIChat

#     load_dotenv()

#     # Get the OpenAI API key from the environment variable
#     api_key = os.getenv("GROQ_API_KEY")

#     # Model
#     model = OpenAIChat(
#         openai_api_base="https://api.groq.com/openai/v1",
#         openai_api_key=api_key,
#         model_name="llama-3.1-70b-versatile",
#         temperature=0.1,
#     )
#     # Initialize the vector database
#     vector_db = AgentRouter()

#     # Define specialized system prompts for each agent
#     DATA_EXTRACTOR_PROMPT = """You are a highly specialized private equity agent focused on data extraction from various documents. Your expertise includes:
#         1. Extracting key financial metrics (revenue, EBITDA, growth rates, etc.) from financial statements and reports
#     2. Identifying and extracting important contract terms from legal documents
#     3. Pulling out relevant market data from industry reports and analyses
#     4. Extracting operational KPIs from management presentations and internal reports
#     5. Identifying and extracting key personnel information from organizational charts and bios
#     Provide accurate, structured data extracted from various document types to support investment analysis."""

#     SUMMARIZER_PROMPT = """You are an expert private equity agent specializing in summarizing complex documents. Your core competencies include:
#     1. Distilling lengthy financial reports into concise executive summaries
#     2. Summarizing legal documents, highlighting key terms and potential risks
#     3. Condensing industry reports to capture essential market trends and competitive dynamics
#     4. Summarizing management presentations to highlight key strategic initiatives and projections
#     5. Creating brief overviews of technical documents, emphasizing critical points for non-technical stakeholders
#     Deliver clear, concise summaries that capture the essence of various documents while highlighting information crucial for investment decisions."""

#     FINANCIAL_ANALYST_PROMPT = """You are a specialized private equity agent focused on financial analysis. Your key responsibilities include:
#     1. Analyzing historical financial statements to identify trends and potential issues
#     2. Evaluating the quality of earnings and potential adjustments to EBITDA
#     3. Assessing working capital requirements and cash flow dynamics
#     4. Analyzing capital structure and debt capacity
#     5. Evaluating financial projections and underlying assumptions
#     Provide thorough, insightful financial analysis to inform investment decisions and valuation."""
#     MARKET_ANALYST_PROMPT = """You are a highly skilled private equity agent specializing in market analysis. Your expertise covers:
#     1. Analyzing industry trends, growth drivers, and potential disruptors
#     2. Evaluating competitive landscape and market positioning
#     3. Assessing market size, segmentation, and growth potential
#     4. Analyzing customer dynamics, including concentration and loyalty
#     5. Identifying potential regulatory or macroeconomic impacts on the market
#     Deliver comprehensive market analysis to assess the attractiveness and risks of potential investments."""

#     OPERATIONAL_ANALYST_PROMPT = """You are an expert private equity agent focused on operational analysis. Your core competencies include:
#     1. Evaluating operational efficiency and identifying improvement opportunities
#     2. Analyzing supply chain and procurement processes
#     3. Assessing sales and marketing effectiveness
#     4. Evaluating IT systems and digital capabilities
#     5. Identifying potential synergies in merger or add-on acquisition scenarios
#     Provide detailed operational analysis to uncover value creation opportunities and potential risks."""

#     # Initialize specialized agents
#     data_extractor_agent = Agent(
#         agent_name="Data-Extractor",
#         system_prompt=DATA_EXTRACTOR_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=True,
#         verbose=True,
#         dynamic_temperature_enabled=True,
#         saved_state_path="data_extractor_agent.json",
#         user_name="pe_firm",
#         retry_attempts=1,
#         context_length=200000,
#         output_type="string",
#     )

#     summarizer_agent = Agent(
#         agent_name="Document-Summarizer",
#         system_prompt=SUMMARIZER_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=True,
#         verbose=True,
#         dynamic_temperature_enabled=True,
#         saved_state_path="summarizer_agent.json",
#         user_name="pe_firm",
#         retry_attempts=1,
#         context_length=200000,
#         output_type="string",
#     )
#     financial_analyst_agent = Agent(
#         agent_name="Financial-Analyst",
#         system_prompt=FINANCIAL_ANALYST_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=True,
#         verbose=True,
#         dynamic_temperature_enabled=True,
#         saved_state_path="financial_analyst_agent.json",
#         user_name="pe_firm",
#         retry_attempts=1,
#         context_length=200000,
#         output_type="string",
#     )

#     market_analyst_agent = Agent(
#         agent_name="Market-Analyst",
#         system_prompt=MARKET_ANALYST_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=True,
#         verbose=True,
#         dynamic_temperature_enabled=True,
#         saved_state_path="market_analyst_agent.json",
#         user_name="pe_firm",
#         retry_attempts=1,
#         context_length=200000,
#         output_type="string",
#     )

#     operational_analyst_agent = Agent(
#         agent_name="Operational-Analyst",
#         system_prompt=OPERATIONAL_ANALYST_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=True,
#         verbose=True,
#         dynamic_temperature_enabled=True,
#         saved_state_path="operational_analyst_agent.json",
#         user_name="pe_firm",
#         retry_attempts=1,
#         context_length=200000,
#         output_type="string",
#     )

#     # Create agents (using the agents from the original code)
#     agents_to_add = [
#         data_extractor_agent,
#         summarizer_agent,
#         financial_analyst_agent,
#         market_analyst_agent,
#         operational_analyst_agent,
#     ]

#     # Add agents to the vector database
#     for agent in agents_to_add:
#         vector_db.add_agent(agent)

#     # Example task
#     task = "Analyze the financial statements of a potential acquisition target and identify key growth drivers."

#     # Find the best agent for the task
#     best_agent = vector_db.find_best_agent(task)

#     if best_agent:
#         logger.info(f"Best agent for the task: {best_agent.name}")
#         # Use the best agent to perform the task
#         result = best_agent.run(task)
#         print(f"Task result: {result}")

#         # Update the agent's history in the database
#         vector_db.update_agent_history(best_agent.name)
#     else:
#         print("No suitable agent found for the task.")

#     # Save the vector database
```