```
import os
from typing import Any, Dict


import requests
import tavily
from dotenv import load_dotenv


from swarms import Agent
from swarm_models import OpenAIChat
from swarms.tools.prebuilt.bing_api import fetch_web_articles_bing_api


load_dotenv()


try:
    from openai import OpenAI


    from swarms import BaseLLM
except ImportError as e:
    raise ImportError(f"Required modules are not available: {e}")




def perplexity_api_key():
    try:
        api_key = os.getenv("PPLX_API_KEY")
        return api_key
    except Exception as e:
        print(f"Error: {e}")



class Perplexity(BaseLLM):
    """
    A class to interact with the Perplexity API using OpenAI's interface.
    """

    def __init__(
        self, api_key: str = perplexity_api_key(), *args, **kwargs
    ):
        """
        Initialize the Perplexity class with an API key.

        Args:
            api_key (str): The API key for authenticating with the OpenAI client.
        """
        super().__init__(*args, **kwargs)
        self.client = OpenAI(
            api_key=api_key,
            base_url="https://api.perplexity.ai",
            *args,
            **kwargs,
        )

    def run(self, task: str, *args, **kwargs):
```

```
"""
Run the model to process the given task.


Args:
    task (str): The task to be performed.


Returns:
    dict: The processed output from the model.
"""
messages = [
    {
        "role": "system",
        "content": (
            "You are an artificial intelligence assistant and you need to "
            "engage in a helpful, detailed, polite conversation with a user."
        ),
    },
    {
        "role": "user",
        "content": task,
    },
]
try:
    response = self.client.chat.completions.create(
        model="llama-3-sonar-large-32k-online",
        messages=messages,
            )
            return response
        except Exception as e:
            raise RuntimeError(f"Error running the model: {e}")


def check_exa_api():
    try:
        api_key = os.getenv("EXA_API_KEY")
        return api_key
    except Exception as e:
        print(f"Error: {e}")


class ExaAgent(BaseLLM):
    """
    A class to interact with the Exa API.
    """

    def __init__(
        self, api_key: str = check_exa_api(), *args, **kwargs
    ):
        """
        Initialize the ExaAgent class with an API key.

        Args:
            api_key (str): The API key for authenticating with the Exa client.
        """
        super().__init__(*args, **kwargs)
        try:
            from exa_py import Exa

            self.exa = Exa(api_key=api_key)
        except ImportError as e:
            raise ImportError(f"Failed to import Exa: {e}")

    def run(self, task: str, *args, **kwargs):
        """
        Run a search query using the Exa API.

        Args:
            task (str): The search query.

        Returns:
            dict: The search results from the Exa API.
        """
        try:
            results = self.exa.search(
                task, use_autoprompt=True, *args, **kwargs
            )
            return results
        except Exception as e:
```

```
        raise RuntimeError(f"Error running the search query: {e}")



class ResearchAgent:
    """
    A class to represent a research agent that uses an LLM to summarize content from various sources.
    """

    def __init__(
        self,
        api_key: str = None,
        output_dir: str = "research_base",
        n_results: int = 2,
        temperature: float = 0.2,
        max_tokens: int = 3500,
    ):
        """
        Initialize the ResearchAgent class with necessary parameters.

        Args:
            api_key (str): The API key for the Bing API.
            output_dir (str): The directory for storing memory outputs. Default is "research_base".
            n_results (int): Number of results to return from the memory. Default is 2.
            temperature (float): The temperature setting for the LLM. Default is 0.2.
            max_tokens (int): The maximum number of tokens for the LLM. Default is 3500.
        """
        self.api_key = api_key

        self.llm = OpenAIChat(
            temperature=temperature, max_tokens=max_tokens
        )
        self.agent = self._initialize_agent()

    def _initialize_agent(self):
        """
        Initialize the agent with the provided parameters and system prompt.

        Returns:
            Agent: An initialized Agent instance.
        """
        research_system_prompt = """
        Research Agent LLM Prompt: Summarizing Sources and Content
             Objective: Your task is to summarize the provided sources and the content within those sources. The goal is to create concise, accurate, and informative summaries that capture the key points of the original content.
        Instructions:
        1. Identify Key Information: ...
        2. Summarize Clearly and Concisely: ...
        3. Preserve Original Meaning: ...
        4. Include Relevant Details: ...
        5. Structure: ...
        """

        return Agent(
            agent_name="Research Agent",
            system_prompt=research_system_prompt,
            llm=self.llm,
            max_loops=1,
            autosave=True,
            dashboard=False,
            # tools=[fetch_web_articles_bing_api],
            verbose=True,
        )

    def run(self, task: str, *args, **kwargs):
        """
        Run the research agent to fetch and summarize web articles related to the task.

        Args:
            task (str): The task or query for the agent to process.

        Returns:
            str: The agent's response after processing the task.
        """
        articles = fetch_web_articles_bing_api(task)
        sources_prompts = "".join([task, articles])
        agent_response = self.agent.run(sources_prompts)
```

```
        return agent_response



def check_tavily_api():
    try:
        api_key = os.getenv("TAVILY_API_KEY")
        return api_key
    except Exception as e:
        print(f"Error: {e}")



class TavilyWrapper:
    """
    A wrapper class for the Tavily API to facilitate searches and retrieve relevant information.
    """


    def __init__(self, api_key: str = check_tavily_api()):
        """
        Initialize the TavilyWrapper with the provided API key.


        Args:
            api_key (str): The API key for authenticating with the Tavily API.
        """
        if not isinstance(api_key, str):
            raise TypeError("API key must be a string")
    self.api_key = api_key
    self.client = self._initialize_client(api_key)

def _initialize_client(self, api_key: str) -> Any:
    """
    Initialize the Tavily client with the provided API key.

    Args:
        api_key (str): The API key for authenticating with the Tavily API.

    Returns:
        TavilyClient: An initialized Tavily client instance.
    """
    try:
        return tavily.TavilyClient(api_key=api_key)
    except Exception as e:
        raise RuntimeError(
            f"Error initializing Tavily client: {e}"
        )

def run(self, task: str) -> Dict[str, Any]:
    """
    Perform a search query using the Tavily API.

    Args:
        task (str): The search query.
        Returns:
            dict: The search results from the Tavily API.
        """
        if not isinstance(task, str):
            raise TypeError("Task must be a string")

        try:
            response = self.client.search(
                query=task, search_depth="advanced"
            )
            return response
        except Exception as e:
            raise RuntimeError(f"Error performing search: {e}")


def you_search_api_key():
    try:
        api_key = os.getenv("YOU_API_KEY")
        return api_key
    except Exception as e:
        print(f"Error: {e}")


class YouSearchAgent:
    """
```

```
A wrapper class for the YDC Index API to facilitate fetching AI snippets based on a query.
"""


def __init__(self, api_key: str = you_search_api_key()):
    """
    Initialize the AISnippetsWrapper with the provided API key.


    Args:
        api_key (str): The API key for authenticating with the YDC Index API.
    """


    self.api_key = api_key


def run(self, task: str) -> Dict[str, Any]:
    """
    Fetch AI snippets for the given query using the YDC Index API.


    Args:
        task (str): The search query.


    Returns:
        dict: The search results from the YDC Index API.
    """
    if not isinstance(task, str):
        raise TypeError("Task must be a string")
    headers = {"X-API-Key": os.getenv("YOU_API_KEY")}
    params = {"query": task}

    try:
        response = requests.get(
            f"https://api.ydc-index.io/rag?query={task}",
            params=params,
            headers=headers,
        )
        return response.json()
    except requests.RequestException as e:
        raise RuntimeError(f"Error fetching AI snippets: {e}")


# task = "What is the swarmms framework"

# # Run all of the agents
# agents = [
#     Perplexity,
#     ExaAgent,
#     # ResearchAgent,
#     TavilyWrapper,
#     # YouSearchAgent,
#     # Brave
# ]
# # Run each agent with the given task

# for agent_class in agents:

#     logger.info(f"Running agent: {agent_class.__name__}")

#     agent = agent_class()

#     response = agent.run(task)

#     print(response)
```