```
import openai
from pydantic import BaseModel
import os
from loguru import logger
from swarm_models.base_llm import BaseLLM
from typing import List


class OpenAIFunctionCaller(BaseLLM):
    """
    A class that represents a caller for OpenAI chat completions.

    Args:
        system_prompt (str): The system prompt to be used in the chat completion.
        model_name (str): The name of the OpenAI model to be used.
        max_tokens (int): The maximum number of tokens in the generated completion.
        temperature (float): The temperature parameter for randomness in the completion.
        base_model (BaseModel): The base model to be used for the completion.
        openai_api_key (str): The API key for accessing the OpenAI service.
        parallel_tool_calls (bool): Whether to make parallel tool calls.
        top_p (float): The top-p parameter for nucleus sampling in the completion.

    Attributes:
        system_prompt (str): The system prompt to be used in the chat completion.
        model_name (str): The name of the OpenAI model to be used.
        max_tokens (int): The maximum number of tokens in the generated completion.
    temperature (float): The temperature parameter for randomness in the completion.

    base_model (BaseModel): The base model to be used for the completion.

    parallel_tool_calls (bool): Whether to make parallel tool calls.

    top_p (float): The top-p parameter for nucleus sampling in the completion.

    client (openai.OpenAI): The OpenAI client for making API calls.


Methods:

    check_api_key: Checks if the API key is provided and retrieves it from the environment if not.

    run: Runs the chat completion with the given task and returns the generated completion.


"""


def __init__(

    self,

    system_prompt: str = None,

    model_name: str = "gpt-4o-2024-08-06",

    max_tokens: int = 4000,

    temperature: float = 0.4,

    base_model: BaseModel = None,

    openai_api_key: str = None,

    parallel_tool_calls: bool = False,

    top_p: float = 0.9,

    *args,

    **kwargs,

):

    super().__init__()
```

```
        self.system_prompt = system_prompt
        self.model_name = model_name
        self.max_tokens = max_tokens
        self.temperature = temperature
        self.openai_api_key = openai_api_key
        self.base_model = base_model
        self.parallel_tool_calls = parallel_tool_calls
        self.top_p = top_p
        self.client = openai.OpenAI(api_key=self.check_api_key())

    def check_api_key(self) -> str:
        """
        Checks if the API key is provided and retrieves it from the environment if not.

        Returns:
            str: The API key.

        """
        if self.openai_api_key is None:
            self.openai_api_key = os.getenv("OPENAI_API_KEY")

        return self.openai_api_key

    def run(self, task: str, *args, **kwargs) -> dict:
        """
        Runs the chat completion with the given task and returns the generated completion.
```

```
Args:
    task (str): The user's task for the chat completion.
    *args: Additional positional arguments to be passed to the OpenAI API.
    **kwargs: Additional keyword arguments to be passed to the OpenAI API.

Returns:
    str: The generated completion.

"""
try:
    completion = self.client.beta.chat.completions.parse(
        model=self.model_name,
        messages=[
            {"role": "system", "content": self.system_prompt},
            {"role": "user", "content": task},
        ],
        max_tokens=self.max_tokens,
        temperature=self.temperature,
        response_format=self.base_model,
        parallel_tool_calls=self.parallel_tool_calls,
        tools=(
            [openai.pydantic_function_tool(self.base_model)]
        ),
        *args,
        **kwargs,
        )

        out = (
            completion.choices[0]
            .message.tool_calls[0]
            .function.arguments
        )

        # Conver str to dict
        # print(out)
        out = eval(out)
        return out
    except Exception as error:
        logger.error(
            f"Error in running OpenAI chat completion: {error}"
        )
        return None

def convert_to_dict_from_base_model(
    self, base_model: BaseModel
) -> dict:
    return openai.pydantic_function_tool(base_model)


def convert_list_of_base_models(
    self, base_models: List[BaseModel]
):
        """
        Converts a list of BaseModels to a list of dictionaries.

        Args:
            base_models (List[BaseModel]): A list of BaseModels to be converted.

        Returns:
            List[Dict]: A list of dictionaries representing the converted BaseModels.
        """
        return [
            self.convert_to_dict_from_base_model(base_model)
            for base_model in base_models
        ]


# def agents_list(
#     agents: List[Agent] = None,
# ) -> str:
#     responses = []

#     for agent in agents:
#         name = agent.agent_name
#         description = agent.description
#         response = f"Agent Name {name}: Description {description}"
#         responses.append(response)
#     return concat_strings(responses)



# class HierarchicalOrderCall(BaseModel):
#     agent_name: str
#     task: str



# # Example usage:
# # Initialize the function caller
# function_caller = OpenAIFunctionCaller(
#     system_prompt="You are a helpful assistant.",
#     openai_api_key="","
#     max_tokens=500,
#     temperature=0.5,
#     base_model=HierarchicalOrderCall,
# )

# # Run the function caller
# response = function_caller.run(
#     "Send an order to the financial agent twice"
# )
# print(response)
```