```
import os
import subprocess
from typing import List, Optional

from loguru import logger
from pydantic import BaseModel, Field
from pydantic.v1 import validator
from swarm_models import OpenAIChat
from tenacity import (
    retry,
    stop_after_attempt,
    wait_exponential,
)

from swarms.structs.agent import Agent
from swarms.structs.swarm_router import SwarmRouter, SwarmType

logger.add("swarm_builder.log", rotation="10 MB", backtrace=True)


class OpenAIFunctionCaller:
    """
    A class to interact with the OpenAI API for generating text based on a system prompt and a task.

    Attributes:
    - system_prompt (str): The system prompt to guide the AI's response.
```

```
- api_key (str): The API key for the OpenAI service.
- temperature (float): The temperature parameter for the AI model, controlling randomness.
- base_model (BaseModel): The Pydantic model to parse the response into.
- max_tokens (int): The maximum number of tokens in the response.
- client (OpenAI): The OpenAI client instance.
"""

def __init__(
    self,
    system_prompt: str,
    api_key: str,
    temperature: float,
    base_model: BaseModel,
    max_tokens: int = 5000,
):
    self.system_prompt = system_prompt
    self.api_key = api_key
    self.temperature = temperature
    self.base_model = base_model
    self.max_tokens = max_tokens

    try:
        from openai import OpenAI
    except ImportError:
        logger.error(
            "OpenAI library not found. Please install the OpenAI library by running 'pip install openai'"
```

```
        )
        subprocess.run(["pip", "install", "openai"])
        from openai import OpenAI

    self.client = OpenAI(api_key=api_key)

def run(self, task: str, *args, **kwargs) -> BaseModel:
    """
    Run the OpenAI model with the system prompt and task to generate a response.

    Args:
    - task (str): The task to be completed.
    - *args: Additional positional arguments for the OpenAI API.
    - **kwargs: Additional keyword arguments for the OpenAI API.

    Returns:
    - BaseModel: The parsed response based on the base_model.
    """
    completion = self.client.beta.chat.completions.parse(
        model="gpt-4o-2024-08-06",
        messages=[
            {"role": "system", "content": self.system_prompt},
            {"role": "user", "content": task},
        ],
        response_format=self.base_model,
        temperature=self.temperature,
```

```
        max_tokens=self.max_tokens,
        *args,
        **kwargs,
    )

    return completion.choices[0].message.parsed

@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10),
)
async def run_async(
    self, task: str, *args, **kwargs
) -> BaseModel:
    """
    Asynchronous version of the run method.

    Args:
    - task (str): The task to be completed.
    - *args: Additional positional arguments for the OpenAI API.
    - **kwargs: Additional keyword arguments for the OpenAI API.

    Returns:
    - BaseModel: The parsed response based on the base_model.
    """
    completion = (
```

```
            await self.client.beta.chat.completions.parse_async(
                model="gpt-4o-2024-08-06",
                messages=[
                    {"role": "system", "content": self.system_prompt},
                    {"role": "user", "content": task},
                ],
                response_format=self.base_model,
                temperature=self.temperature,
                max_tokens=self.max_tokens,
                *args,
                **kwargs,
            )
        )

        return completion.choices[0].message.parsed


BOSS_SYSTEM_PROMPT = """
Manage a swarm of worker agents to efficiently serve the user by deciding whether to create new agents or delegate tasks. Ensure operations are efficient and effective.

### Instructions:

1. **Task Assignment**:
   - Analyze available worker agents when a task is presented.
   - Delegate tasks to existing agents with clear, direct, and actionable instructions if an appropriate
```

agent is available.

- If no suitable agent exists, create a new agent with a fitting system prompt to handle the task.

2. **Agent Creation**:

- Name agents according to the task they are intended to perform (e.g., "Twitter Marketing Agent").

- Provide each new agent with a concise and clear system prompt that includes its role, objectives, and any tools it can utilize.

3. **Efficiency**:

- Minimize redundancy and maximize task completion speed.

- Avoid unnecessary agent creation if an existing agent can fulfill the task.

4. **Communication**:

- Be explicit in task delegation instructions to avoid ambiguity and ensure effective task execution.

- Require agents to report back on task completion or encountered issues.

5. **Reasoning and Decisions**:

- Offer brief reasoning when selecting or creating agents to maintain transparency.

- Avoid using an agent if unnecessary, with a clear explanation if no agents are suitable for a task.

Output Format

Present your plan in clear, bullet-point format or short concise paragraphs, outlining task assignment, agent creation, efficiency strategies, and communication protocols.

Notes

- Preserve transparency by always providing reasoning for task-agent assignments and creation.
- Ensure instructions to agents are unambiguous to minimize error.

"""

```
class AgentConfig(BaseModel):
    """Configuration for an individual agent in a swarm"""

    name: str = Field(
        description="The name of the agent",
    )
    description: str = Field(
        description="A description of the agent's purpose and capabilities",
    )
    system_prompt: str = Field(
        description="The system prompt that defines the agent's behavior",
    )


class SwarmConfig(BaseModel):
    """Configuration for a swarm of cooperative agents"""

    name: str = Field(
```

```
        description="The name of the swarm",
        example="Research-Writing-Swarm",
    )
    description: str = Field(
        description="The description of the swarm's purpose and capabilities",
        example="A swarm of agents that work together to research topics and write articles",
    )
    agents: List[AgentConfig] = Field(
        description="The list of agents that make up the swarm",
    )
    max_loops: int = Field(
        description="The maximum number of loops for the swarm to iterate on",
    )

    @validator("agents")
    def validate_agents(cls, v):
        if not v:
            raise ValueError("Swarm must have at least one agent")
        return v


class AutoSwarmBuilderOutput(BaseModel):
    """A class that automatically builds and manages swarms of AI agents with enhanced error handling."""

    name: Optional[str] = Field(
```

```
    description="The name of the swarm",
    example="DefaultSwarm",
    default=None,
)
description: Optional[str] = Field(
    description="The description of the swarm's purpose and capabilities",
    example="Generic AI Agent Swarm",
    default=None,
)
verbose: Optional[bool] = Field(
    description="Whether to display verbose output",
    default=None,
)
model_name: Optional[str] = Field(
    description="The name of the OpenAI model to use",
    default=None,
)
boss_output_schema: Optional[list] = Field(
    description="The schema for the output of the BOSS system prompt",
    default=None,
)
director_agents_created: Optional[SwarmConfig] = Field(
    description="The agents created by the director",
    default=None,
)
swarm_router_outputs: Optional[list] = Field(
```

```
        description="The outputs from the swarm router",
        default=None,
    )
    max_loops: Optional[int] = Field(
        description="The maximum number of loops for the swarm to iterate on",
        default=None,
    )
    swarm_type: Optional[SwarmType] = Field(
        description="The type of swarm to build",
        default=None,
    )


class AutoSwarmBuilder:
    """A class that automatically builds and manages swarms of AI agents with enhanced error handling."""

    def __init__(
        self,
        name: Optional[str] = "autonomous-swarm-builder",
        description: Optional[
            str
        ] = "Given a task, this swarm will automatically create specialized agents and route it to the appropriate agents.",
        verbose: bool = True,
        model_name: str = "gpt-4o",
```

```
    boss_output_schema: list = None,
    swarm_router_outputs: AutoSwarmBuilderOutput = None,
    max_loops: int = 1,
    swarm_type: str = "SequentialWorkflow",
    auto_generate_prompts_for_agents: bool = False,
    shared_memory_system: callable = None,
):
    self.name = name or "DefaultSwarm"
    self.description = description or "Generic AI Agent Swarm"
    self.verbose = verbose
    self.agents_pool = []
    self.api_key = os.getenv("OPENAI_API_KEY")
    self.model_name = model_name
    self.boss_output_schema = boss_output_schema
    self.max_loops = max_loops
    self.swarm_type = swarm_type
    self.auto_generate_prompts_for_agents = (
        auto_generate_prompts_for_agents
    )
    self.shared_memory_system = shared_memory_system
    self.auto_swarm_builder_output = AutoSwarmBuilderOutput(
        name=name,
        description=description,
        verbose=verbose,
        model_name=model_name,
        boss_output_schema=boss_output_schema or [],
```

```
    swarm_router_outputs=swarm_router_outputs or [],
    max_loops=max_loops,
    swarm_type=swarm_type,
)

if not self.api_key:
    raise ValueError(
        "OpenAI API key must be provided either through initialization or environment variable"
    )

logger.info(
    "Initialized AutoSwarmBuilder",
    extra={
        "swarm_name": self.name,
        "description": self.description,
        "model": self.model_name,
    },
)

# Initialize OpenAI chat model
try:
    self.chat_model = OpenAIChat(
        openai_api_key=self.api_key,
        model_name=self.model_name,
    )
except Exception as e:
```

```
        logger.error(
            f"Failed to initialize OpenAI chat model: {str(e)}"
        )
        raise


def run(
    self,
    task: str,
    image_url: Optional[str] = None,
    *args,
    **kwargs,
):
    """Run the swarm on a given task with error handling and retries."""
    if not task or not task.strip():
        raise ValueError("Task cannot be empty")

    logger.info("Starting swarm execution", extra={"task": task})

    try:
        # Create agents for the task
        agents = self._create_agents(task)
        if not agents:
            raise ValueError(
                "No agents were created for the task"
            )
```

```
    # Execute the task through the swarm router
    logger.info(
        "Routing task through swarm",
        extra={"num_agents": len(agents)},
    )
    output = self.swarm_router(
        agents=agents,
        task=task,
        image_url=image_url,
        *args,
        **kwargs,
    )
    self.auto_swarm_builder_output.swarm_router_outputs.append(
        output
    )
    print(output)

    logger.info("Swarm execution completed successfully")
    # return output
    return self.auto_swarm_builder_output.model_dump_json(
        indent=4
    )

except Exception as e:
    logger.error(
        f"Error during swarm execution: {str(e)}",
```

```
        )
        raise e


def _create_agents(
    self,
    task: str,
) -> List[Agent]:
    """Create the necessary agents for a task with enhanced error handling."""
    logger.info("Creating agents for task", extra={"task": task})

    try:
        model = OpenAIFunctionCaller(
            system_prompt=BOSS_SYSTEM_PROMPT,
            api_key=self.api_key,
            temperature=0.1,
            base_model=SwarmConfig,
        )

        agents_config = model.run(task)
        logger.info(
            f"Director has successfully created agents: {agents_config}"
        )
        self.auto_swarm_builder_output.director_agents_created = (
            agents_config
        )
```

```
if isinstance(agents_config, dict):
    agents_config = SwarmConfig(**agents_config)

# Update swarm configuration
self.name = agents_config.name
self.description = agents_config.description

# Create agents from configuration
agents = []
for agent_config in agents_config.agents:
    if isinstance(agent_config, dict):
        agent_config = AgentConfig(**agent_config)

    agent = self.build_agent(
        agent_name=agent_config.name,
        agent_description=agent_config.description,
        agent_system_prompt=agent_config.system_prompt,
    )
    agents.append(agent)

    print(
        f"Agent created: {agent_config.name}: Description: {agent_config.description}"
    )

# # Add available agents showcase to system prompts
# agents_available = showcase_available_agents(
```

```
        #     name=self.name,
        #     description=self.description,
        #     agents=agents,
        # )

        # for agent in agents:
        #     agent.system_prompt += "\n" + agents_available

        logger.info(
            "Successfully created agents",
            extra={"num_agents": len(agents)},
        )
        return agents

    except Exception as e:
        logger.error(
            f"Error creating agents: {str(e)}", exc_info=True
        )
        raise

def build_agent(
    self,
    agent_name: str,
    agent_description: str,
    agent_system_prompt: str,
    *args,
```

```
    **kwargs,
) -> Agent:
    """Build a single agent with enhanced error handling."""
    logger.info(
        "Building agent", extra={"agent_name": agent_name}
    )

    try:
        agent = Agent(
            agent_name=agent_name,
            description=agent_description,
            system_prompt=agent_system_prompt,
            llm=self.chat_model,
            verbose=self.verbose,
            dynamic_temperature_enabled=False,
            return_step_meta=False,
            output_type="str",
            streaming_on=True,
        )
        return agent

    except Exception as e:
        logger.error(
            f"Error building agent: {str(e)}", exc_info=True
        )
        raise
```

```
@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10),
)
def swarm_router(
    self,
    agents: List[Agent],
    task: str,
    img: Optional[str] = None,
    *args,
    **kwargs,
) -> str:
    """Route tasks between agents in the swarm with error handling and retries."""
    logger.info(
        "Initializing swarm router",
        extra={"num_agents": len(agents)},
    )

    try:
        swarm_router_instance = SwarmRouter(
            name=self.name,
            description=self.description,
            agents=agents,
            swarm_type=self.swarm_type,
            auto_generate_prompts=self.auto_generate_prompts_for_agents,
```

```
            )

            # formatted_task = f"{self.name} {self.description} {task}"
            result = swarm_router_instance.run(
                task=task, *args, **kwargs
            )

            logger.info("Successfully completed swarm routing")
            return result

        except Exception as e:
            logger.error(
                f"Error in swarm router: {str(e)}", exc_info=True
            )
            raise


swarm = AutoSwarmBuilder(
    name="ChipDesign-Swarm",
    description="A swarm of specialized AI agents for chip design",
    swarm_type="ConcurrentWorkflow",
)

try:
    result = swarm.run(
        "Design a new AI accelerator chip optimized for transformer model inference..."
```

```
    )
    print(result)
except Exception as e:
    print(f"An error occurred: {e}")
```