```
import os
import time
from typing import List, Dict, Any, Union
from pydantic import BaseModel, Field
from loguru import logger
from swarms import Agent
from swarm_models import OpenAIChat
from dotenv import load_dotenv

# Load environment variables
load_dotenv()


# Pydantic model to track metadata for each agent
class AgentMetadata(BaseModel):
    agent_id: str
    start_time: float = Field(default_factory=time.time)
    end_time: Union[float, None] = None
    task: str
    output: Union[str, None] = None
    error: Union[str, None] = None
    status: str = "running"


# Worker Agent class
class WorkerAgent:
```

```
def __init__(
    self,
    agent_name: str,
    system_prompt: str,
    model_name: str = "gpt-4o-mini",
):
    """Initialize a Worker agent with its own model, name, and system prompt."""
    api_key = os.getenv("OPENAI_API_KEY")

    # Create the LLM model for the worker
    self.model = OpenAIChat(
        openai_api_key=api_key,
        model_name=model_name,
        temperature=0.1,
    )

    # Initialize the worker agent with a unique prompt and name
    self.agent = Agent(
        agent_name=agent_name,
        system_prompt=system_prompt,
        llm=self.model,
        max_loops=1,
        autosave=True,
        dashboard=False,
        verbose=True,
        dynamic_temperature_enabled=True,
```

```
        saved_state_path=f"{agent_name}_state.json",
        user_name="swarms_corp",
        retry_attempts=1,
        context_length=200000,
        return_step_meta=False,
    )

def perform_task(self, task: str) -> Dict[str, Any]:
    """Perform the task assigned by the Queen and return the result."""
    metadata = AgentMetadata(
        agent_id=self.agent.agent_name, task=task
    )

    try:
        logger.info(
            f"{self.agent.agent_name} is starting task '{task}'."
        )
        result = self.agent.run(task)
        metadata.output = result
        metadata.status = "completed"
    except Exception as e:
        logger.error(
            f"{self.agent.agent_name} encountered an error: {e}"
        )
        metadata.error = str(e)
        metadata.status = "failed"
```

```
        finally:
            metadata.end_time = time.time()
            return metadata.dict()


# Queen Agent class to manage the workers and dynamically decompose tasks
class QueenAgent:
    def __init__(
        self,
        worker_count: int = 5,
        model_name: str = "gpt-4o-mini",
        queen_name: str = "Queen-Agent",
        queen_prompt: str = "You are the queen of the hive",
    ):
        """Initialize the Queen agent who assigns tasks to workers.

        Args:
            worker_count (int): Number of worker agents to manage.
            model_name (str): The model used by worker agents.
            queen_name (str): The name of the Queen agent.
            queen_prompt (str): The system prompt for the Queen agent.
        """
        self.queen_name = queen_name
        self.queen_prompt = queen_prompt

        # Queen agent initialization with a unique prompt for dynamic task decomposition
```

```
        api_key = os.getenv("OPENAI_API_KEY")
        self.queen_model = OpenAIChat(
            openai_api_key=api_key,
            model_name=model_name,
            temperature=0.1,
        )

        # Initialize worker agents
        self.workers = [
            WorkerAgent(
                agent_name=f"Worker-{i+1}",
                system_prompt=f"Worker agent {i+1}, specialized in helping with financial analysis.",
                model_name=model_name,
            )
            for i in range(worker_count)
        ]
        self.worker_metadata: Dict[str, AgentMetadata] = {}

    def decompose_task(self, task: str) -> List[str]:
        """Dynamically decompose a task into multiple subtasks using prompting."""
        decomposition_prompt = f"""You are a highly efficient problem solver. Given the following task:
          '{task}', please decompose this task into 3-5 smaller subtasks, and explain how they can be completed step by step."""

        logger.info(
            f"{self.queen_name} is generating subtasks using prompting for the task: '{task}'"
```

```
    )

    # Use the queen's model to generate subtasks dynamically
    subtasks_output = self.queen_model.run(decomposition_prompt)
    logger.info(f"Queen output: {subtasks_output}")
    subtasks = subtasks_output.split("\n")

    # Filter and clean up subtasks
    subtasks = [
        subtask.strip() for subtask in subtasks if subtask.strip()
    ]

    return subtasks

def assign_subtasks(self, subtasks: List[str]) -> Dict[str, Any]:
    """Assign subtasks to workers dynamically and collect their results.

    Args:
        subtasks (List[str]): The list of subtasks to distribute among workers.

    Returns:
        dict: A dictionary containing results from workers.
    """
    logger.info(
        f"{self.queen_name} is assigning subtasks to workers."
    )
```

```
        results = {}

        for i, subtask in enumerate(subtasks):
            # Assign each subtask to a different worker
            worker = self.workers[
                i % len(self.workers)
            ]  # Circular assignment if more subtasks than workers
            worker_result = worker.perform_task(subtask)
            results[worker_result["agent_id"]] = worker_result
            self.worker_metadata[worker_result["agent_id"]] = (
                worker_result
            )

        return results

    def gather_results(self) -> Dict[str, Any]:
        """Gather all results from the worker agents."""
        return self.worker_metadata

    def run_swarm(self, task: str) -> Dict[str, Any]:
            """Run the swarm by decomposing a task into subtasks, assigning them to workers, and gathering results."""
        logger.info(f"{self.queen_name} is initializing the swarm.")

        # Decompose the task into subtasks using prompting
        subtasks = self.decompose_task(task)
```

```
        logger.info(f"Subtasks generated by the Queen: {subtasks}")

        # Assign subtasks to workers
        results = self.assign_subtasks(subtasks)

        logger.info(
            f"{self.queen_name} has collected results from all workers."
        )
        return results


# Example usage
if __name__ == "__main__":
    # Queen oversees 3 worker agents with a custom system prompt
    queen = QueenAgent(
        worker_count=3,
        queen_name="Queen-Overseer",
          queen_prompt="You are the overseer queen of a financial analysis swarm. Decompose and distribute tasks wisely.",
    )

    # Task for the swarm to execute
    task = "Analyze the best strategies to establish a ROTH IRA and maximize tax savings."

    # Run the swarm on the task and gather results
    final_results = queen.run_swarm(task)
```

```
print("Final Swarm Results:", final_results)
```