```
import os
from typing import Any, Callable, Dict, List, Optional, Tuple, Union

import yaml
from tenacity import (
    retry,
    stop_after_attempt,
    wait_exponential,
    retry_if_exception_type,
)
from pydantic import (
    BaseModel,
    Field,
    field_validator,
)
from swarms.utils.loguru_logger import initialize_logger
from swarms.structs.agent import Agent
from swarms.structs.swarm_router import SwarmRouter
from swarms.utils.litellm_wrapper import LiteLLM

logger = initialize_logger(log_folder="create_agents_from_yaml")


class AgentConfig(BaseModel):
    agent_name: str
    system_prompt: str
```

```
model_name: Optional[str] = None

max_loops: int = Field(default=1, ge=1)

autosave: bool = True

dashboard: bool = False

verbose: bool = False

dynamic_temperature_enabled: bool = False

saved_state_path: Optional[str] = None

user_name: str = "default_user"

retry_attempts: int = Field(default=3, ge=1)

context_length: int = Field(default=100000, ge=1000)

return_step_meta: bool = False

output_type: str = "str"

auto_generate_prompt: bool = False

artifacts_on: bool = False

artifacts_file_extension: str = ".md"

artifacts_output_path: str = ""


@field_validator("system_prompt")

@classmethod

def validate_system_prompt(cls, v):

    if not v or not isinstance(v, str) or len(v.strip()) == 0:

        raise ValueError(

            "System prompt must be a non-empty string"

        )

    return v
```

```
class SwarmConfig(BaseModel):
    name: str
    description: str
    max_loops: int = Field(default=1, ge=1)
    swarm_type: str
    task: Optional[str] = None
    flow: Optional[Dict] = None
    autosave: bool = True
    return_json: bool = False
    rules: str = ""

    @field_validator("swarm_type")
    @classmethod
    def validate_swarm_type(cls, v):
        valid_types = {
            "SequentialWorkflow",
            "ConcurrentWorkflow",
            "AgentRearrange",
            "MixtureOfAgents",
            "auto",
        }
        if v not in valid_types:
            raise ValueError(
                f"Swarm type must be one of: {valid_types}"
            )
```

```
        return v



class YAMLConfig(BaseModel):
    agents: List[AgentConfig] = Field(..., min_length=1)
    swarm_architecture: Optional[SwarmConfig] = None

    model_config = {
        "extra": "forbid"  # Prevent additional fields not in the model
    }



def load_yaml_safely(
    yaml_file: str = None, yaml_string: str = None
) -> Dict:
    """Safely load and validate YAML configuration using Pydantic."""
    try:
        if yaml_string:
            config_dict = yaml.safe_load(yaml_string)
        elif yaml_file:
            if not os.path.exists(yaml_file):
                raise FileNotFoundError(
                    f"YAML file {yaml_file} not found."
                )
            with open(yaml_file, "r") as file:
                config_dict = yaml.safe_load(file)
```

```
        else:
            raise ValueError(
                "Either yaml_file or yaml_string must be provided"
            )

        # Validate using Pydantic
        YAMLConfig(**config_dict)
        return config_dict
    except yaml.YAMLError as e:
        raise ValueError(f"Error parsing YAML: {str(e)}")
    except Exception as e:
        raise ValueError(f"Error validating configuration: {str(e)}")



@retry(
    stop=stop_after_attempt(3),
    wait=wait_exponential(multiplier=1, min=4, max=10),
    retry=retry_if_exception_type((ConnectionError, TimeoutError)),
    before_sleep=lambda retry_state: logger.info(
        f"Retrying after error: {retry_state.outcome.exception()}"
    ),
)
def create_agent_with_retry(
    agent_config: Dict, model: LiteLLM
) -> Agent:
    """Create an agent with retry logic for handling transient failures."""
```

```
try:
    validated_config = AgentConfig(**agent_config)
    agent = Agent(
        agent_name=validated_config.agent_name,
        system_prompt=validated_config.system_prompt,
        llm=model,
        max_loops=validated_config.max_loops,
        autosave=validated_config.autosave,
        dashboard=validated_config.dashboard,
        verbose=validated_config.verbose,
        dynamic_temperature_enabled=validated_config.dynamic_temperature_enabled,
        saved_state_path=validated_config.saved_state_path,
        user_name=validated_config.user_name,
        retry_attempts=validated_config.retry_attempts,
        context_length=validated_config.context_length,
        return_step_meta=validated_config.return_step_meta,
        output_type=validated_config.output_type,
        auto_generate_prompt=validated_config.auto_generate_prompt,
        artifacts_on=validated_config.artifacts_on,
        artifacts_file_extension=validated_config.artifacts_file_extension,
        artifacts_output_path=validated_config.artifacts_output_path,
    )
    return agent
except Exception as e:
    logger.error(
        f"Error creating agent {agent_config.get('agent_name', 'unknown')}: {str(e)}"
```

```
        )
        raise


def create_agents_from_yaml(
    model: Callable = None,
    yaml_file: str = "agents.yaml",
    yaml_string: str = None,
    return_type: str = "auto",
) -> Union[
    SwarmRouter,
    Agent,
    List[Agent],
    Tuple[Union[SwarmRouter, Agent], List[Agent]],
    List[Dict[str, Any]],
]:
    """
    Create agents and/or SwarmRouter based on configurations defined in a YAML file or string.
    """
    agents = []
    task_results = []
    swarm_router = None

    try:
        # Load and validate configuration
        config = load_yaml_safely(yaml_file, yaml_string)
```

```
# Create agents with retry logic
for agent_config in config["agents"]:
    logger.info(
        f"Creating agent: {agent_config['agent_name']}"
    )

    if "model_name" in agent_config:
        model_instance = LiteLLM(
            model_name=agent_config["model_name"]
        )
    else:
        model_name = "gpt-4o"
        model_instance = LiteLLM(model_name=model_name)

    agent = create_agent_with_retry(
        agent_config, model_instance
    )
    logger.info(
        f"Agent {agent_config['agent_name']} created successfully."
    )
    agents.append(agent)

# Create SwarmRouter if specified
if "swarm_architecture" in config:
    try:
```

```
        swarm_config = SwarmConfig(
            **config["swarm_architecture"]
        )
        swarm_router = SwarmRouter(
            name=swarm_config.name,
            description=swarm_config.description,
            max_loops=swarm_config.max_loops,
            agents=agents,
            swarm_type=swarm_config.swarm_type,
            task=swarm_config.task,
            flow=swarm_config.flow,
            autosave=swarm_config.autosave,
            return_json=swarm_config.return_json,
            rules=swarm_config.rules,
        )
        logger.info(
            f"SwarmRouter '{swarm_config.name}' created successfully."
        )
    except Exception as e:
        logger.error(f"Error creating SwarmRouter: {str(e)}")
        raise ValueError(
            f"Failed to create SwarmRouter: {str(e)}"
        )

# Handle return types with improved error checking
valid_return_types = {
```

```
    "auto",
    "swarm",
    "agents",
    "both",
    "tasks",
    "run_swarm",
}
if return_type not in valid_return_types:
    raise ValueError(
        f"Invalid return_type. Must be one of: {valid_return_types}"
    )

if return_type == "run_swarm" or "swarm":
    if not swarm_router:
        raise ValueError(
            "Cannot run swarm: SwarmRouter not created."
        )
    try:
        return swarm_router.run(
            config["swarm_architecture"]["task"]
        )
    except Exception as e:
        logger.error(f"Error running SwarmRouter: {str(e)}")
        raise

# Return appropriate type based on configuration
```

```
    if return_type == "auto":
        return (
            swarm_router
            if swarm_router
            else (agents[0] if len(agents) == 1 else agents)
        )
    elif return_type == "swarm":
        return (
            swarm_router
            if swarm_router
            else (agents[0] if len(agents) == 1 else agents)
        )
    elif return_type == "agents":
        return agents[0] if len(agents) == 1 else agents
    elif return_type == "both":
        return (
            swarm_router
            if swarm_router
            else agents[0] if len(agents) == 1 else agents
        ), agents
    elif return_type == "tasks":
        return task_results

except Exception as e:
    logger.error(
        f"Critical error in create_agents_from_yaml: {str(e)}"
```

)

raise