```
import json
import os
from contextlib import suppress
from typing import Any, Callable, Dict, Optional, Type, Union

from dotenv import load_dotenv
from pydantic import BaseModel, Field, ValidationError, create_model
from swarm_models.openai_function_caller import OpenAIFunctionCaller



class DynamicParser:
    @staticmethod
    def extract_fields(model: Type[BaseModel]) -> Dict[str, Any]:
        return {
            field_name: (
                field.annotation,
                ... if field.is_required() else None,
            )
            for field_name, field in model.model_fields.items()
        }

    @staticmethod
    def create_partial_model(
        model: Type[BaseModel], data: Dict[str, Any]
    ) -> Type[BaseModel]:
        fields = {
```

```
        field_name: (
            field.annotation,
            ... if field.is_required() else None,
        )
        for field_name, field in model.model_fields.items()
        if field_name in data
    }
    return create_model(f"Partial{model.__name__}", **fields)

@classmethod
def parse(
    cls, data: Union[str, Dict[str, Any]], model: Type[BaseModel]
) -> Optional[BaseModel]:
    if isinstance(data, str):
        try:
            data = json.loads(data)
        except json.JSONDecodeError:
            return None

    # Try full model first
    with suppress(ValidationError):
        return model.model_validate(data)

    # Create and try partial model
    partial_model = cls.create_partial_model(model, data)
    with suppress(ValidationError):
```

```
            return partial_model.model_validate(data)


        return None



load_dotenv()



# Define the Thoughts schema
class Thoughts(BaseModel):
    text: str = Field(
        ...,
        description="Current thoughts or observations regarding the task.",
    )
    reasoning: str = Field(
        ...,
        description="Logical reasoning behind the thought process.",
    )
    plan: str = Field(
        ...,
        description="A short bulleted list that conveys the immediate and long-term plan.",
    )
    criticism: str = Field(
        ...,
        description="Constructive self-criticism to improve future responses.",
    )
```

```
    speak: str = Field(
        ...,
        description="A concise summary of thoughts intended for the user.",
    )


# Define the Command schema
class Command(BaseModel):
    name: str = Field(
        ...,
        description="Command name to execute from the provided list of commands.",
    )
    args: Dict[str, Any] = Field(
        ..., description="Arguments required to execute the command."
    )


# Define the AgentResponse schema
class AgentResponse(BaseModel):
    thoughts: Thoughts = Field(
        ..., description="The agent's current thoughts and reasoning."
    )
    command: Command = Field(
        ...,
        description="The command to execute along with its arguments.",
    )
```

```
# Define tool functions
def fluid_api_command(task: str):
    """Execute a fluid API request."""
    # response = fluid_api_request(task)
    print(response.model_dump_json(indent=4))
    return response


def send_tweet_command(text: str):
    """Simulate sending a tweet."""
    print(f"Tweet sent: {text}")
    return {"status": "success", "message": f"Tweet sent: {text}"}


def do_nothing_command():
    """Do nothing."""
    print("Doing nothing...")
    return {"status": "success", "message": "No action taken."}


def task_complete_command(reason: str):
    """Mark the task as complete and provide a reason."""
    print(f"Task completed: {reason}")
    return {
```

```
        "status": "success",
        "message": f"Task completed: {reason}",
    }


# Dynamic command execution
def execute_command(name: str, args: Dict[str, Any]):
    """Dynamically execute a command based on its name and arguments."""
    command_map: Dict[str, Callable] = {
        "fluid_api": lambda **kwargs: fluid_api_command(
            task=kwargs.get("task")
        ),
        "send_tweet": lambda **kwargs: send_tweet_command(
            text=kwargs.get("text")
        ),
        "do_nothing": lambda **kwargs: do_nothing_command(),
        "task_complete": lambda **kwargs: task_complete_command(
            reason=kwargs.get("reason")
        ),
    }

    if name not in command_map:
        raise ValueError(f"Unknown command: {name}")

    # Execute the command with the provided arguments
    return command_map[name](**args)
```

```
def parse_and_execute_command(
    response: Union[str, Dict[str, Any]],
    base_model: Type[BaseModel] = AgentResponse,
) -> Any:
    """Enhanced command parser with flexible input handling"""
    parsed = DynamicParser.parse(response, base_model)
    if not parsed:
        raise ValueError("Failed to parse response")

    if hasattr(parsed, "command"):
        command_name = parsed.command.name
        command_args = parsed.command.args
        return execute_command(command_name, command_args)

    return parsed



ainame = "AutoAgent"
userprovided = "assistant"

SYSTEM_PROMPT = f"""
You are {ainame}, an advanced and autonomous {userprovided}.
Your role is to make decisions and complete tasks independently without seeking user assistance.
Leverage your strengths as an LLM to solve tasks efficiently, adhering strictly to the commands and
```

resources provided.

GOALS:

1. {userprovided}

2. Execute tasks with precision and efficiency.

3. Ensure outputs are actionable and aligned with the user's objectives.

4. Continuously optimize task strategies for maximum effectiveness.

5. Maintain reliability and consistency in all responses.

CONSTRAINTS:

1. Memory limit: ~4000 words for short-term memory. Save essential information to files immediately to avoid loss.

2. Independent decision-making: Do not rely on user assistance.

3. Exclusively use commands in double quotes (e.g., "command name").

4. Use subprocesses for commands that may take longer than a few minutes.

5. Ensure all outputs strictly adhere to the specified JSON response format.

COMMANDS:

1. Fluid API: "fluid_api", args: "method": "<GET/POST/...>", "url": "<url>", "headers": "<headers>", "body": "<payload>"

18. Send Tweet: "send_tweet", args: "text": "<text>"

19. Do Nothing: "do_nothing", args:

20. Task Complete (Shutdown): "task_complete", args: "reason": "<reason>"

RESOURCES:

1. Internet access for real-time information and data gathering.

2. Long-term memory management for storing critical information.

3. Access to GPT-3.5-powered Agents for delegating tasks.

4. File handling capabilities for output storage and retrieval.

PERFORMANCE EVALUATION:

1. Continuously analyze and reflect on actions to ensure optimal task completion.

2. Self-critique decisions and strategies constructively to identify areas for improvement.

3. Ensure every command serves a clear purpose and minimizes resource usage.

4. Complete tasks in the least number of steps, balancing speed and accuracy.

RESPONSE FORMAT:

Always respond in a strict JSON format as described below. Ensure your responses can be parsed with Python's `json.loads`:

"""

```
# Initialize the OpenAIFunctionCaller
model = OpenAIFunctionCaller(
    system_prompt=SYSTEM_PROMPT,
    max_tokens=4000,
    temperature=0.9,
    base_model=AgentResponse,  # Pass the Pydantic schema as the base model
    parallel_tool_calls=False,
    openai_api_key=os.getenv("OPENAI_API_KEY"),
)

# Example usage
```

```
user_input = (
    "Analyze the provided Python code for inefficiencies, generate suggestions for improvements, "
    "and provide optimized code."
)

response = model.run(user_input)
response = parse_and_execute_command(response)
print(response)
```