Browser_use as function call

```
from langchain_openai import ChatOpenAI
from browser_use import Agent
import asyncio

async def main():
    agent = Agent(
        task="Find a one-way flight from Bali to Oman on 12 January 2025 on Google Flights.
Return me the cheapest option.",
        llm=ChatOpenAI(model="gpt-4o"),
    )
    result = await agent.run()
    print(result)

asyncio.run(main())
```

action.py

```
import litellm
from typing import Dict, Union

from ..utils.config import Config


class ActionConfig(Config):
        required_fields = [
        "agent_name",
        "agent_role",
        "is_user",
        "is_node_begin",
        "is_agent_begin",
        "used_prompt_templates",
        "prompts_order",
        "system_prompt",
        "last_prompt",
        "tools_results_dict",
        "history_messages",
        "response",
        "content",
        "latency",
        "start_time_ms",
        "end_time_ms",
        ]

        def __init__(self, config_path_or_dict: Union[str, dict] = None) -> None:
        super().__init__(config_path_or_dict)
        self._validate_config()
        self.agent_name: str = self.config_dict["agent_name"]
        self.agent_role: str = self.config_dict["agent_role"]
        self.is_user: bool = self.config_dict["is_user"]
        self.is_node_begin: bool = self.config_dict["is_node_begin"]
        self.is_agent_begin: bool = self.config_dict["is_agent_begin"]
        self.used_prompt_templates: dict = self.config_dict["used_prompt_templates"]
        self.prompts_order: list = self.config_dict["prompts_order"]
        self.system_prompt: str = self.config_dict["system_prompt"]
        self.last_prompt: str = self.config_dict["last_prompt"]
        self.tools_results_dict: Dict[str, dict] = self.config_dict["tools_results_dict"]
        self.history_messages: list = self.config_dict["history_messages"]
        self.response: Union[litellm.utils.ModelResponse, dict] =
self.config_dict["response"]
        self.content: str = self.config_dict["content"]
        self.latency: float = self.config_dict["latency"]
        self.start_time_ms: int = self.config_dict["start_time_ms"]
        self.end_time_ms: int = self.config_dict["end_time_ms"]


class Action:
        """
        The basic action unit of agent
        """

        def __init__(self, config: ActionConfig):
        self.config = config

        # agent相关
        self.agent_name = self.config.agent_name
        self.agent_role = self.config.agent_role
        self.is_user = self.config.is_user

        # node相关
        self.is_node_begin = self.config.is_node_begin
        self.is_agent_begin = self.config.is_agent_begin

        # 输入输出相关
        self.used_prompt_templates = self.config.used_prompt_templates
        self.prompts_order = self.config.prompts_order
        self.system_prompt = self.config.system_prompt
        self.last_prompt = self.config.last_prompt
        self.tools_results_dict = self.config.tools_results_dict
        self.history_messages = self.config.history_messages

        # prompt输入大模型后生成的回复，是一个response对象，但是序列化为json后此项
信息全部保留到了response_json中
        self.response = self.config.response
```

```
# response对象中的content，即大模型生成的回复的字符串对象
self.content = self.config.content

if isinstance(self.response, litellm.utils.ModelResponse):
# response对象的json格式
self.response_json = self.response.json()
# prompt中的token数量
self.token_usage = self.response_json.get("usage")
else:
self.response_json = None
self.token_usage = None

# 时间相关
self.latency = self.config.latency
self.start_time_ms = self.config.start_time_ms
self.end_time_ms = self.config.end_time_ms

def to_dict(self):
return {
"agent_name": self.agent_name,
"agent_role": self.agent_role,
"is_user": self.is_user,
"is_node_begin": self.is_node_begin,
"is_agent_begin": self.is_agent_begin,
"content": self.content,
"used_prompt_templates": self.used_prompt_templates,
"prompts_order": self.prompts_order,
"system_prompt": self.system_prompt,
"last_prompt": self.last_prompt,
"history_messages": self.history_messages,
"tools_results_dict": self.tools_results_dict,
"response_json": self.response_json,
"token_usage": self.token_usage,
"latency": self.latency,
"start_time_ms": self.start_time_ms,
"end_time_ms": self.end_time_ms,
}
```

```
agent.py
import json
import logging
import litellm
from datetime import datetime
from typing import Dict, Union

from .llm import LLMConfig, OpenAILLM
from .memory import Memory, ShortTermMemory, LongTermMemory
from .toolkit import Toolkit
from .environment import Environment
from .action import ActionConfig, Action
from ..tools import Tool
from ..utils.prompts import *
from ..utils.config import Config
from ..utils.files import save_logs

logger = logging.getLogger(__name__)


class AgentConfig(Config):

        required_fields = ["agent_name", "agent_roles"]

        def __init__(self, config_path_or_dict: Union[str, dict] = None) -> None:
        super().__init__(config_path_or_dict)
        self._validate_config()

        self.agent_name: str = self.config_dict["agent_name"]
        self.agent_roles: Dict[str, str] = self.config_dict["agent_roles"]
        self.agent_style: str = self.config_dict.get("agent_style")
        self.agent_description: str = self.config_dict.get("agent_description")
        self.LLM_config: dict = self.config_dict.get("LLM_config")
        self.toolkit: dict = self.config_dict.get("toolkit")
        self.memory: dict = self.config_dict.get("memory")
        self.is_user: bool = self.config_dict.get("is_user", False)

        @classmethod
        def generate_config(cls, agent_name: str, agent_roles: Dict[str, str]):
        llm_config = LLMConfig()
        llm_config.log_path = f"logs/{agent_name}"

        return cls(
        config_path_or_dict={
                "agent_name": agent_name,
                "agent_roles": agent_roles,
                "LLM_config": llm_config.to_dict(),
        }
        )


class Agent:
        def __init__(self, config: AgentConfig):
        self.config = config
        self.agent_name = self.config.agent_name
        self.agent_roles = self.config.agent_roles
        self.agent_style = self.config.agent_style
        self.agent_description = self.config.agent_description
        llm_config = (
        LLMConfig(self.config.LLM_config) if self.config.LLM_config else None
        )
        self.LLM = OpenAILLM(llm_config) if llm_config else None
        if self.config.memory:
        self.short_term_memory = (
                ShortTermMemory(
                config=self.config.memory["short_term_memory"], messages=[]
                )
                if "short_term_memory" in self.config.memory
                else ShortTermMemory(config={}, messages=[])
        )
        self.long_term_memory = (
                LongTermMemory(
                config=self.config.memory["long_term_memory"],
                json_path=self.config.memory["long_term_memory"].get(
                "json_path", f"memory/{self.agent_name}.jsonl"
                ),
                chunk_list=[],
                )
                if "long_term_memory" in self.config.memory
                else LongTermMemory(
                config={},
                json_path=f"memory/{self.agent_name}.jsonl",
                chunk_list=[],
                )
        )
        else:
        self.short_term_memory = ShortTermMemory(config={}, messages=[])
        self.long_term_memory = LongTermMemory(
                config={},
                json_path=f"memory/{self.agent_name}.jsonl",
                chunk_list=[],
        )
        self.toolkit = (
        Toolkit.from_config(self.config.toolkit) if self.config.toolkit else None
        )
        self.is_user = self.config.is_user
        def observe(self, environment: Environment):
        """
        observe the environment and return the observation
        Return: observation(dict)
        """
        # If the environment type is cooperative, the agent will share the information
with other agents
        if environment.environment_type == "cooperative":
        short_term_memory_messages = environment.shared_memory[
                "short_term_memory"
        ].get_memory()
        environment_relevant_memory = Memory.encode_memory(
                short_term_memory_messages, self.agent_name
        )
        # Otherwise, the agent will not use the shared memory
        else:
        environment_relevant_memory = "None."

        agent_relevant_memory = self.short_term_memory.get_memory_string(
        self.agent_name
        )

        observation = OBSERVATION_TEMPLATE.format(
        environment_relevant_memory=environment_relevant_memory,
        agent_relevant_memory=agent_relevant_memory,
        )
        observation = {
        "role": "user",
        "content": observation,
        }
        return observation

        def step(self, current_node, environment: Environment, user_input=None):
        """
        return actions by current state and environment
        Return: action(Action)
        """

        is_node_begin = current_node.is_begin
        current_node.is_begin = False
        is_agent_begin = (
        True
        if is_node_begin
        and current_node.begin_role
        and current_node.begin_role ==
current_node.name_role_hash[self.agent_name]
        else False
        )

        used_prompt_templates = {}
        prompts_order = []
        response = {}
        content = ""
        tools_results_dict = {}
        system_prompt = ""
        last_prompt = ""
        history_messages = []

        # If the agent is acted by the user, then the agent will use the user input as the
content
        start_time = None
        end_time = None
        if self.is_user:
        user_input = user_input if user_input else input(f"{self.agent_name}: ")
        content = user_input
        # Otherwise the agent will use LLM to generate the response
        else:
        # First update the information according to the current environment
        if len(environment.shared_memory["short_term_memory"].get_memory()) > 0:
                observation = self.observe(environment)
                if observation:
                self.short_term_memory.append_memory(observation)

        # If the agent is the first to speak in the node, then the agent will use the
predefined begin_query
        if is_agent_begin and current_node.begin_query:
                content = current_node.begin_query
        # Otherwise, the agent will use the LLM to generate the response
        else:
                start_time = datetime.now().timestamp()
                history_messages = self.short_term_memory.get_memory()
                system_prompt, last_prompt = self.compile(current_node)

                agent_role = current_node.name_role_hash[self.agent_name]
                if agent_role in current_node.node_primary_prompts:
                prompts_order.extend(
                list(current_node.node_primary_prompts[agent_role].keys())
                )

                if agent_role in current_node.node_prompt_paddings:
                for prompt_type in
current_node.node_prompt_paddings[agent_role].keys():
                if prompt_type not in DEFAULT_NODE_PROMPT_TEMPLATES:
                        prompts_order.append(prompt_type)
                        used_prompt_templates[prompt_type] = (
                    current_node.node_prompt_templates[prompt_type]
                    )

                available_tools = []
                if self.toolkit and self.toolkit.tool_specifications:
                available_tools.extend(self.toolkit.tool_specifications)
                if (
                environment.shared_toolkit
                and environment.shared_toolkit.tool_specifications
                ):
                available_tools.extend(
                environment.shared_toolkit.tool_specifications
                )
                if current_node.kb and current_node.kb.kb_specification:
                available_tools.append(current_node.kb.kb_specification)
                if len(available_tools) == 0:
                available_tools = None

                response, content = self.LLM.get_response(
                chat_messages=history_messages,
                system_prompt=system_prompt,
                last_prompt=last_prompt,
                stream=False,
                tools=available_tools,
                )
                end_time = datetime.now().timestamp()

        if isinstance(content, litellm.Message) and content.get("tool_calls"):
                tool_calls = content.tool_calls
                for tool_call in tool_calls:
                tool_name = tool_call.function.name
                tool_arguments = json.loads(tool_call.function.arguments)
                # If the tool is the knowledge_base_retriever, then the agent will retrieve
the information from the knowledge base
                if tool_name == "knowledge_base_retriever":
                print("\nRetrieving from knowledge base...\n")
                retrieved_context = current_node.kb.retrieve(**tool_arguments)
                content = retrieved_context
                # If the tool is in the toolkit, then the agent will call the tool to generate
the content
                else:
                if self.toolkit and tool_name in self.toolkit.tools:
                    tool: Tool = self.toolkit.tools[tool_name]
                elif (
                    environment.shared_toolkit
                    and tool_name in environment.shared_toolkit.tools
                ):
                    tool: Tool = environment.shared_toolkit.tools[tool_name]
                else:
                    raise ValueError(
                    f"Tool {tool_name} is not found in the toolkit."
                    )
                tool_result = tool.func(**tool_arguments)
                tools_results_dict[tool_call.id] = tool_result
                content = tool_result["content"]
                save_logs(self.LLM.log_path, history_messages, content)
        else:
                content = content.strip(f"Speak content:").strip()

        print(f"\n{self.agent_name}: {content}\n")

        action_dict = {
        "used_prompt_templates": used_prompt_templates,
        "prompts_order": prompts_order,
        "response": response,
        "content": content,
        "tools_results_dict": tools_results_dict,
        "agent_role": current_node.name_role_hash[self.agent_name],
        "agent_name": self.agent_name,
        "is_user": self.is_user,
        "is_node_begin": is_node_begin,
        "is_agent_begin": is_agent_begin,
        "system_prompt": system_prompt,
        "last_prompt": last_prompt,
        "history_messages": history_messages,
        "latency": end_time - start_time if start_time is not None else 0,
        "start_time_ms": round(start_time * 1000) if start_time is not None else 0,
        "end_time_ms": round(end_time * 1000) if end_time is not None else 0,
        }
        action = Action(config=ActionConfig(action_dict))
        return action

        def compile(self, current_node):
        """
        get prompt from state depend on your role
        Return:
        system_prompt:system_prompt for agents's LLM
        last_prompt:last_prompt for agents's LLM
        """
        system_prompt: str = current_node.node_description
        last_prompt: str = ""
        if self.agent_name in current_node.node_prompts:
        for prompt_type, prompt in current_node.node_prompts[
                self.agent_name
        ].items():
                if prompt:
            last_prompt = last_prompt + "\n" + prompt
        last_prompt = AGENT_LAST_PROMPT_TEMPLATE.format(
        last_prompt=last_prompt,
        name=self.agent_name,
        )

        return system_prompt, last_prompt

        def to_dict(self, node_name: str):
        """used for serialization in state"""
        # attention: the agent_role should be the role of the agent in the current node,
        # not the role of the agent in the whole task."""

        return {
        "agent_name": self.agent_name,
        "agent_roles": self.agent_roles,
        "agent_role": self.agent_roles.get(node_name, None),
        "agent_style": self.agent_style,
        "agent_description": self.agent_description,
        "is_user": self.is_user,
        "toolkit": self.config.toolkit,
        "LLM": self.config.LLM_config if self.LLM else None,
        "long_term_memory": self.long_term_memory.get_memory(),
        "short_term_memory": self.short_term_memory.get_memory(),
        }
```

Agent_team.py

```
mport os
import copy
import json
from typing import Dict, Union

from .agent import AgentConfig, Agent
from .llm import OpenAILLM, LLMConfig
from .memory import ShortTermMemory, LongTermMemory
from .environment import EnvironmentConfig, Environment
from .action import Action
from ..utils.config import Config
from ..utils.prompts import AGENT_TEAM_CONFIG_GENERATION_PROMPT_TEMPLATE


class AgentTeamConfig(Config):

        required_fields = ["agents", "environment"]

        def __init__(self, config_path_or_dict: Union[str, dict] = None) -> None:
        super().__init__(config_path_or_dict)
        self._validate_config()

        self.agents: dict = self.config_dict["agents"]
        self.environment: dict = self.config_dict["environment"]

        @classmethod
        def generate_config(
        cls, task_description: str, all_node_roles_description: Dict[str, dict]
        ):
        llm_config = {
        "LLM_type": "OpenAI",
        "model": "gpt-4-turbo-2024-04-09",
        "temperature": 0.3,
        "log_path": "logs/generate_config/agent_team",
        "ACTIVE_MODE": True,
        "SAVE_LOGS": True,
        }

        all_roles_description = ""
        for node_name, node_roles_description in all_node_roles_description.items():
        all_roles_description += f"Node '{node_name}' contains the roles
'{list(node_roles_description.keys())}', they cannot be allocated to the same people. "
        for role_name, role_description in node_roles_description.items():
                all_roles_description += (
                "At node '{node_name}', {role_name} {role_description}. ".format(
                node_name=node_name,
                role_name=role_name,
                role_description=role_description[0].lower()
                + role_description[1:].strip("."),
                )
                )

        llm = OpenAILLM(LLMConfig(llm_config))
        system_prompt = "You are a helpful assistant designed to output JSON."
        last_prompt =
AGENT_TEAM_CONFIG_GENERATION_PROMPT_TEMPLATE.format(
        task_description=task_description,
        all_roles_description=all_roles_description.strip(),
        )

        response, content = llm.get_response(
        chat_messages=None,
        system_prompt=system_prompt,
        last_prompt=last_prompt,
        )

        # Converting the JSON format string to a JSON object
        json_config = json.loads(content.strip("`").strip("json").strip())
        agents_dict = {}
        for agent_name, agent_roles in json_config.items():
        agent_config = AgentConfig.generate_config(agent_name, agent_roles)
        agents_dict[agent_name] = agent_config.to_dict()

        environment_config = EnvironmentConfig()

        return cls(
        config_path_or_dict={
                "agents": agents_dict,
                "environment": environment_config.to_dict(),
        }
        )


class AgentTeam:
        """Agent has a team of agents, and the team has an environment.
        The team is responsible for the interaction between agents and the environment."""

        def __init__(self, config: AgentTeamConfig):
        self.config = config
        self.agents: Dict[str, Agent] = {}
        if self.config.agents:
        for agent_config in self.config.agents.values():
                agent = Agent(config=AgentConfig(agent_config))
                self.agents[agent.agent_name] = agent
        self.environment: Environment = (
        Environment(config=EnvironmentConfig(self.config.environment))
        if self.config.environment
        else None
        )

        def step(self, agent_name, current_node, user_input=None):
        return self.agents[agent_name].step(
        current_node=current_node,
        environment=self.environment,
        user_input=user_input,
        )

        def execute(self, action: Action):
        content = ""
        for res in action.content:
        content += res

        # Delete the third person from the dialogue
        parse = "{action.agent_name}:"
        content = content.replace(parse, "")
        # Update short-term memory in shared memory of the environment
        shared_short_term_memory: ShortTermMemory = self.environment.shared_memory[
        "short_term_memory"
        ]
        shared_short_term_memory.append_memory(
        {
                "name": action.agent_name,
                "role": action.agent_role,
                "content": content,
        }
        )

        # Update summary in shared memory of the environment
        ENVIRONMENT_SUMMARY_STEP = eval(
        os.environ.get("ENVIRONMENT_SUMMARY_STEP", "10")
        )
        if len(shared_short_term_memory) % ENVIRONMENT_SUMMARY_STEP == 0:
        summary = shared_short_term_memory.get_memory_summary()
        self.environment.shared_memory["summary"] = summary

        # Update long-term memory in shared memory of the environment
        shared_long_term_memory: LongTermMemory = self.environment.shared_memory[
        "long_term_memory"
        ]
        shared_long_term_memory.append_memory_from_short_term_memory(
        shared_short_term_memory
        )

        # Update short-term memory and long-term memory of the act agent if it is not acted
by the user
        if not self.agents[action.agent_name].is_user:
        self.agents[action.agent_name].short_term_memory.append_memory(
                {
                "name": action.agent_name,
                "role": "assistant",
                "content": content,
                }
        )
        self.agents[
                action.agent_name
        ].long_term_memory.append_memory_from_short_term_memory(
                self.agents[action.agent_name].short_term_memory
        )

        def dump(self, path):
        save_config = copy.deepcopy(self.config)
        # todo 优化agent和environment时再修改，此时不涉及对agent的更新

        save_config.dump(path)
```

llm.py

```
import os
import time
from abc import abstractmethod
from typing import Union

import litellm
from dotenv import load_dotenv

from ..utils.config import Config
from ..utils.files import save_logs

load_dotenv()

WAIT_TIME = 20


# @backoff.on_exception(backoff.expo, litellm.OpenAIError, max_tries=100)
def completion_with_backoff(**kwargs):
        litellm.api_key = os.environ["OPENAI_API_KEY"]
        litellm.api_base = os.environ.get("OPENAI_BASE_URL")

        if os.environ.get("OPENAI_API_KEY") is None:
        raise ValueError("OPENAI_API_KEY is not set")

        while True:
        try:
        return litellm.completion(**kwargs)
        except litellm.OpenAIError:
        print(f"Please wait {WAIT_TIME} seconds and resend later ...")
        time.sleep(WAIT_TIME)


class LLMConfig(Config):

        required_fields = []

        def __init__(self, config_path_or_dict: Union[str, dict] = None):
        super().__init__(config_path_or_dict)
        self._validate_config()

        self.LLM_type: str = self.config_dict.get("LLM_type", "OpenAI")
        self.model: str = self.config_dict.get("model", "gpt-4-turbo-2024-04-09")
        self.temperature: float = self.config_dict.get("temperature", 0.3)
        self.log_path: str = self.config_dict.get("log_path", "logs")
        self.API_KEY: str = self.config_dict.get(
        "OPENAI_API_KEY", os.environ["OPENAI_API_KEY"]
        )
        self.API_BASE = self.config_dict.get(
        "OPENAI_BASE_URL", os.environ.get("OPENAI_BASE_URL")
        )
        self.MAX_CHAT_MESSAGES: int = self.config_dict.get("max_chat_messages", 10)
        self.ACTIVE_MODE: bool = self.config_dict.get("ACTIVE_MODE", False)
        self.SAVE_LOGS: bool = self.config_dict.get("SAVE_LOGS", False)


class LLM:
        def __init__(self, config: LLMConfig) -> None:
        self.config = config
        self.model = self.config.model
        self.temperature = self.config.temperature
        self.log_path = self.config.log_path
        self.API_KEY = self.config.API_KEY
        self.API_BASE = self.config.API_BASE
        self.MAX_CHAT_MESSAGES = self.config.MAX_CHAT_MESSAGES
        self.ACTIVE_MODE = self.config.ACTIVE_MODE
        self.SAVE_LOGS = self.config.SAVE_LOGS

        @abstractmethod
        def get_response(cls, **kwargs):
        pass


class OpenAILLM(LLM):

        def __init__(self, config: LLMConfig) -> None:
        super().__init__(config)
        assert self.config.LLM_type == "OpenAI"

        def get_stream(self, response, log_path, messages):
        ans = ""
        for res in response:
        if res:
                r = (
                res.choices[0]["delta"].get("content")
                if res.choices[0]["delta"].get("content")
                else ""
                )
                ans += r
                yield r

        if self.SAVE_LOGS:
```

```
save_logs(log_path, messages, ans)

def get_response(
self,
chat_messages,
system_prompt,
last_prompt=None,
stream=False,
tools=None,
tool_choice="auto",
response_format=None,
**kwargs,
):
"""
return LLM's response
"""
litellm.api_key = self.API_KEY
if self.API_BASE:
litellm.api_base = self.API_BASE

messages = (
[{"role": "system", "content": system_prompt}] if system_prompt else []
)

if chat_messages:
if len(chat_messages) > self.MAX_CHAT_MESSAGES:
        chat_messages = chat_messages[-self.MAX_CHAT_MESSAGES :]
if isinstance(chat_messages[0], dict):
        messages += chat_messages

if last_prompt:
if self.ACTIVE_MODE:
        last_prompt += " Be accurate but concise in response."

if messages:
        # messages[-1]["content"] += last_prompt
        messages.append({"role": "user", "content": last_prompt})
else:
        messages = [{"role": "user", "content": last_prompt}]

if tools:
response = completion_with_backoff(
        model=self.model,
        messages=messages,
        tools=tools,
        tool_choice=tool_choice,
        temperature=self.temperature,
        response_format=response_format,
        custom_llm_provider="openai",
)
else:
response = completion_with_backoff(
        model=self.model,
        messages=messages,
        temperature=self.temperature,
        stream=stream,
        response_format=response_format,
        custom_llm_provider="openai",
)

if response.choices[0].message.get("tool_calls"):
content = response.choices[0].message
elif stream:
content = self.get_stream(response, self.log_path, messages)
else:
content = response.choices[0].message["content"].strip()
if self.SAVE_LOGS:
        save_logs(self.log_path, messages, content)

return response, content
```

memory.py

```
import os
import torch
from pathlib import Path
from abc import ABC, abstractmethod
from typing import Optional
from text2vec import semantic_search

from .llm import completion_with_backoff
from ..utils.storages import (
        InMemoryKeyValueStorage,
        JsonStorage,
)
from ..utils.embeddings import get_embedding
from ..utils.prompts import *


class Memory(ABC):
        def __init__(self) -> None:
        pass

        @abstractmethod
        def get_memory():
        pass
```

```
    @staticmethod
    def encode_memory(massages: list, agent_name=None):
    """Convert a sequence of messages to strings and encode them into one string."""
    encoded_memory = ""
    for message in massages:
    name = message.get("name")
    if agent_name and name and agent_name == name:
            name = f"you({agent_name})"
    role, content = message["role"], message["content"]
    single_message = SINGLE_MESSAGE_TEMPLATE.format(
            name=name, role=role, content=content
    )
    encoded_memory += "\n" + single_message
    return encoded_memory

    @staticmethod
    def get_relevant_memory(query: str, history: list, embeddings: torch.Tensor):
    """
    Retrieve a list of key history entries based on a query using semantic search.

    Args:
    query (str): The input query for which key history is to be retrieved.
    history (list): A list of historical key entries.
    embeddings (numpy.ndarray): An array of embedding vectors for historical entries.

    Returns:
    list: A list of key history entries most similar to the query.
    """
    top_k = eval(os.environ["TOP_K"]) if "TOP_K" in os.environ else 5
    relevant_memory = []
    query_embedding = get_embedding(query)
    hits = semantic_search(
    query_embedding, embeddings, top_k=min(top_k, embeddings.shape[0])
    )
    hits = hits[0]
    for hit in hits:
    matching_idx = hit["corpus_id"]
    try:
            relevant_memory.append(history[matching_idx])
    except:
            return []
    return relevant_memory


class ShortTermMemory(Memory):
    """An implementation of the :obj:`Memory` abstract base class for
    maintaining a record of chat histories.
```

```
Args:
config (dict): A dictionary containing configuration
storage (BaseKeyValueStorage, optional): A storage mechanism for
storing chat history. If `None`, an :obj:`InMemoryKeyValueStorage`
will be used. (default: :obj:`None`)
window_size (int, optional): Specifies the number of recent chat
messages to retrieve. If not provided, the entire chat history
will be retrieved. (default: :obj:`None`)
"""

def __init__(
self,
config: dict = {},
messages: list = [],
storage: InMemoryKeyValueStorage = None,
window_size: Optional[int] = None,
):
self.config = config
self.storage = storage or InMemoryKeyValueStorage()
self.storage.save(
messages
)  # list of gpt-format messages e.g.  [{"name": "" , "role": "", "content": ""}]
self.window_size = window_size

def __len__(self):
return len(self.storage)

def get_memory(self):
return self.storage.load()

def append_memory(self, message):
self.storage.save([message])

def update_memory(self, config, messages):
self.config = config
self.storage.clear()
self.storage.save(messages)

def get_memory_string(self, agent_name=None):
encoded_memory = Memory.encode_memory(self.get_memory(), agent_name)
return encoded_memory

def get_memory_embedding(self):
encoded_memory = self.get_memory_string()
embed = get_embedding(encoded_memory)
return embed

def get_memory_summary(self):
        encoded_memory = self.get_memory_string()

        # Using GPT-3.5 to summarize the conversation
        response = completion_with_backoff(
        model="gpt-3.5-turbo-0125",
        messages=[
                {"role": "system", "content": "You are a helpful assistant."},
                {
                "role": "user",
                "content": SUMMARY_PROMPT_TEMPLATE.format(
                conversation=encoded_memory
                ),
                },
        ],
        temperature=0,
        )
        summary = response.choices[0].message["content"].strip()
        return summary

        def to_dict(self):
        return {
        "config": self.config,
        "memory": self.get_memory(),
        }

        @staticmethod
        def load_from_json(json_dict):
        return ShortTermMemory(
        config=json_dict["config"],
        messages=json_dict["memory"],
        )


class LongTermMemory(Memory):

        def __init__(
        self,
        config: dict,
        json_path: str,
        chunk_list: list = [],
        storage: JsonStorage = None,
        window_size: Optional[int] = 3,
        ):
        self.config = config
        self.json_path = json_path
        if storage:
        self.storage = storage
        else:
```

```
        os.makedirs(os.path.dirname(json_path), exist_ok=True)
        self.storage = JsonStorage(Path(json_path))
        self.storage.save(chunk_list)
        self.window_size = window_size

        def get_memory(self):
        return self.storage.load()

        def update_memory(self, config, chunk_list):
        self.config = config
        self.storage.clear()
        self.storage.save(chunk_list)

        def append_memory(self, chunk):
        self.storage.save([chunk])

        def append_memory_from_short_term_memory(self, short_term_memory:
ShortTermMemory):
        if len(short_term_memory) >= self.window_size:
        memory = short_term_memory.get_memory()
        self.storage.save([Memory.encode_memory(memory[-self.window_size :])])

        def to_dict(self):
        return {
        "config": self.config,
        "memory": self.get_memory(),
        }

        @staticmethod
        def load_from_json(json_dict):
        return LongTermMemory(
        config=json_dict["config"],
        chunk_list=json_dict["memory"],
        )
toolkit.py
import json
from typing import Dict, List

from ..tools import Tool, AVAILABLE_TOOLS


class Toolkit:
        def __init__(self, config: dict, **kwargs):
        self.config = config
        self.tools: Dict[str, Tool] = kwargs.get("tools", {})
        self.tool_specifications: List[dict] = kwargs.get("tool_specifications", None)

        @classmethod
```

```
    def from_config(cls, config_path_or_dict):
    if isinstance(config_path_or_dict, str):
    with open(config_path_or_dict, encoding="utf-8") as f:
        config = json.load(f)
    elif isinstance(config_path_or_dict, dict):
    config = config_path_or_dict
    else:
    raise ValueError("config_path_or_dict should be a path or a dict")

    tools = {}
    tool_specifications = []
    for tool_name, tool_config in config.items():
    if tool_name in AVAILABLE_TOOLS:
        tool: Tool = AVAILABLE_TOOLS[tool_name](**tool_config)
        tools[tool_name] = tool
        tool_specifications.append(
        {
        "type": tool.type,
        "function": {
            "name": tool.name,
            "description": tool.description,
            "parameters": tool.parameters,
        },
        }
        )
    else:
        raise ValueError(
        f"Tool {tool_name} is not available, the available tools are
{AVAILABLE_TOOLS.keys()}"
        )
    if len(tool_specifications) == 0:
    tool_specifications = None

    toolkit = cls(
    config=config, tools=tools, tool_specifications=tool_specifications
    )
    return toolkit

    def to_dict(self):
    return {
    "tools": {
        tool_name: tool.to_dict() for tool_name, tool in self.tools.items()
    },
    "tool_specifications": self.tool_specifications,
    }

    def generate_config():
    # generate the config (especially the prompts)
```

```
        Pass

environment.py
from .memory import ShortTermMemory, LongTermMemory
from .toolkit import Toolkit
from typing import Dict, Union

from ..utils.config import Config


class EnvironmentConfig(Config):

        required_fields = []

        def __init__(self, config_path_or_dict: Union[str, dict] = None) -> None:
        super().__init__(config_path_or_dict)
        self._validate_config()

        self.environment_type: str = self.config_dict.get(
        "environment_type", "cooperative"
        )
        assert self.environment_type in ["cooperative", "competitive"]
        self.shared_memory: Dict[str, dict] = self.config_dict.get(
        "shared_memory", None
        )
        self.shared_toolkit: dict = self.config_dict.get("shared_toolkit", None)


class Environment:
        def __init__(self, config: EnvironmentConfig):
        self.config = config
        self.environment_type = self.config.environment_type
        if self.config.shared_memory:
        short_term_memory = ShortTermMemory(
                config=self.config.shared_memory.get("short_term_memory", {}),
                messages=[],
        )
        long_term_memory = LongTermMemory(
                config=self.config.shared_memory.get("long_term_memory", {}),
                json_path=f"memory/environment.jsonl",
                chunk_list=[],
        )
        self.shared_memory = {
                "summary": self.config.shared_memory.get("summary", ""),
                "short_term_memory": short_term_memory,
                "long_term_memory": long_term_memory,
        }
        else:
```

```
self.shared_memory = {
        "summary": "",
        "short_term_memory": ShortTermMemory(config={}, messages=[]),
        "long_term_memory": LongTermMemory(
        config={}, json_path=f"memory/environment.jsonl", chunk_list=[]
        ),
}
self.shared_toolkit: Toolkit = (
Toolkit.from_config(self.config.shared_toolkit)
if self.config.shared_toolkit
else None
)

def summary(self):
pass

def to_dict(self):
# FIXME: Environment to_dict error
return {
"environment_type": self.environment_type,
"shared_memory": {
        "summary": self.shared_memory["summary"],
        "short_term_memory": self.shared_memory["short_term_memory"].to_dict(),
        "long_term_memory": self.shared_memory["long_term_memory"].to_dict(),
},
"shared_toolkit": self.shared_toolkit.to_dict(),
}

@staticmethod
def load_from_json(json_data):
# FIXME: Environment load from json error
# environment type 和 config通过原始加载方法就能导入
loaded_environment = Environment.from_config(json_data["config"])

# 加载具体的memory
loaded_environment.shared_memory["summary"] = json_data["shared_memory"][
"summary"
]
loaded_environment.shared_memory["short_term_memory"] = (
ShortTermMemory.load_from_json(
        json_data["shared_memory"]["short_term_memory"]
)
)
loaded_environment.shared_memory["long_term_memory"] = (
LongTermMemory.load_from_json(
        json_data["shared_memory"]["long_term_memory"]
)
)
```

```
        return loaded_environment
```

Tools

https://github.com/harishsg993010/HawkinsRAG/

Use Hawkins-Rag as function call
Hawkins-rag

```
from hawkins_rag import HawkinsRAG

# Initialize RAG system
rag = HawkinsRAG()

# Load and process a document
result = rag.load_document("document.txt", source_type="text")

# Query your content
response = rag.query("What is this document about?")
print(response)
```

Browser_use as function call

```
from langchain_openai import ChatOpenAI
from browser_use import Agent
import asyncio

async def main():
   agent = Agent(
      task="Find a one-way flight from Bali to Oman on 12 January 2025 on Google Flights.
Return me the cheapest option.",
      llm=ChatOpenAI(model="gpt-4o"),
   )
   result = await agent.run()
   print(result)

asyncio.run(main())
```

Use Hawkins-Rag as knowledge base

```
from hawkins_rag import HawkinsRAG

# Initialize RAG system
rag = HawkinsRAG()

# Load and process a document
result = rag.load_document("document.txt", source_type="text")
```

```
# Query your content
response = rag.query("What is this document about?")
print(response)
```

Code_interpreter.py

```
import os
from interpreter import OpenInterpreter

from .tool import Tool


class CodeInterpreterTool(Tool):

    def __init__(self, model: str = "gpt-4", api_key: str = None, api_base: str = None):
    description = "Order a programmer to write and run code based on the description of
a problem."
    name = "code_interpreter"
    parameters = {
    "type": "object",
    "properties": {
        "query": {
        "type": "string",
        "description": "The description of the problem.",
        }
    },
    "required": ["query"],
    }
    super().__init__(description, name, parameters)
    self.interpreter = OpenInterpreter()
    self.interpreter.llm.model = model
    self.interpreter.llm.api_key = (
    api_key if api_key else os.environ["OPENAI_API_KEY"]
    )
    if api_base:
    self.interpreter.llm.api_base = api_base
    elif "OPENAI_BASE_URL" in os.environ:
    self.interpreter.llm.api_base = os.environ["OPENAI_BASE_URL"]

    def func(self, query: str):
    messages = self.interpreter.chat(query, display=False)

    code = []
    console = []
    content = ""
    for message in messages:
    if message["type"] == "code":
```

```
                code.append(message["content"])
        elif message["type"] == "console":
                console.append(message["content"])
        elif message["type"] == "message":
                content += message["content"] + "\n"

        return {
        "messages": messages,
        "code": code,
        "console": console,
        "content": content,
        }
```

mail.py

```
import os
import re
import base64
from tqdm import tqdm
from typing import List, Dict
from datetime import datetime, timedelta
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from google.auth.transport.requests import Request
from google.oauth2.credentials import Credentials
from google_auth_oauthlib.flow import InstalledAppFlow
from googleapiclient.discovery import build
from googleapiclient.errors import HttpError

from .tool import Tool


class MailTool(Tool):
        __VALID_ACTION__ = ["read", "send"]

        def __init__(self, cfg_file: str, default_action: str = "read"):
        description = "Read and send e-mails"
        name = "mail"
        parameters = {
        "action": {
                "type": "string",
                "description": "The action to be performed, read or send",
        },
        "state": {
                "type": "string",
                "description": "The state of the email, all, unread, read, sent",
        },
        "time_between": {
                "type": "array",
                    "description": "The time range of the email, such as ['2021/01/01',
'2021/01/02']",
          },
          "sender_mail": {
                    "type": "string",
                    "description": "The sender's email address",
          },
          "only_both": {
                    "type": "boolean",
                    "description": "Whether to search for both sender and recipient",
          },
          "order_by_time": {
                    "type": "string",
                    "description": "The order of the email, descend or ascend",
          },
          "include_word": {
                    "type": "string",
                    "description": "The word to be included in the email",
          },
          "exclude_word": {
                    "type": "string",
                    "description": "The word to be excluded in the email",
          },
          "MAX_SEARCH_CNT": {
                    "type": "integer",
                    "description": "The maximum number of emails to search",
          },
          "number": {
                    "type": "integer",
                    "description": "The number of emails to be returned",
          },
          "recipient_mail": {
                    "type": "string",
                    "description": "The recipient's email address",
          },
          "subject": {
                    "type": "string",
                    "description": "The subject of the email",
          },
          "body": {
                    "type": "string",
                    "description": "The body of the email",
          },
          }
          super(MailTool, self).__init__(description, name, parameters)
          assert (
          default_action.lower() in self.__VALID_ACTION__
```

```
        ), f"Action `{default_action}` is not allowed! The valid action is in
`{self.__VALID_ACTION__}`"
        self.action = default_action.lower()
        self.credential = self._login(cfg_file)

        def _login(self, cfg_file: str):
        SCOPES = [
        "https://www.googleapis.com/auth/gmail.readonly",
        "https://www.googleapis.com/auth/gmail.send",
        ]
        creds = None
        if os.path.exists("token.json"):
        print("Login Successfully!")
        creds = Credentials.from_authorized_user_file("token.json", SCOPES)
        if not creds or not creds.valid:
        print("Please authorize in an open browser.")
        if creds and creds.expired and creds.refresh_token:
                creds.refresh(Request())
        else:
                flow = InstalledAppFlow.from_client_secrets_file(cfg_file, SCOPES)
                creds = flow.run_local_server(port=0)
        # Save the credentials for the next run
        with open("token.json", "w", encoding="utf-8") as token:
                token.write(creds.to_json())
        return creds

        def _read(self, mail_dict: dict):
        credential = self.credential
        state = mail_dict["state"] if "state" in mail_dict else None
        time_between = (
        mail_dict["time_between"] if "time_between" in mail_dict else None
        )
        sender_mail = mail_dict["sender_mail"] if "sender_mail" in mail_dict else None
        only_both = mail_dict["only_both"] if "only_both" in mail_dict else False
        order_by_time = (
        mail_dict["order_by_time"] if "order_by_time" in mail_dict else "descend"
        )
        include_word = (
        mail_dict["include_word"] if "include_word" in mail_dict else None
        )
        exclude_word = (
        mail_dict["exclude_word"] if "exclude_word" in mail_dict else None
        )
        MAX_SEARCH_CNT = (
        mail_dict["MAX_SEARCH_CNT"] if "MAX_SEARCH_CNT" in mail_dict else 50
        )
        number = mail_dict["number"] if "number" in mail_dict else 10
        if state is None:
```

```
        state = "all"
        if time_between is not None:
        assert isinstance(time_between, tuple)
        assert len(time_between) == 2
        assert state in ["all", "unread", "read", "sent"]
        if only_both:
        assert sender_mail is not None
        if sender_mail is not None:
        assert isinstance(sender_mail, str)
        assert credential
        assert order_by_time in ["descend", "ascend"]

        def generate_query():
        query = ""
        if state in ["unread", "read"]:
                query = f"is:{state}"
        if state in ["sent"]:
                query = f"in:{state}"
        if only_both:
                query = f"{query} from:{sender_mail} OR to:{sender_mail}"
        if sender_mail is not None and not only_both:
                query = f"{query} from:({sender_mail})"
        if include_word is not None:
                query = f"{query} {include_word}"
        if exclude_word is not None:
                query = f"{query} -{exclude_word}"
        if time_between is not None:
                TIME_FORMAT = "%Y/%m/%d"
                t1, t2 = time_between
                if t1 == "now":
                t1 = datetime.now().strftime(TIME_FORMAT)
                if t2 == "now":
                t2 = datetime.now().strftime(TIME_FORMAT)
                if isinstance(t1, str) and isinstance(t2, str):
                t1 = datetime.strptime(t1, TIME_FORMAT)
                t2 = datetime.strptime(t2, TIME_FORMAT)
                elif isinstance(t1, str) and isinstance(t2, int):
                t1 = datetime.strptime(t1, TIME_FORMAT)
                t2 = t1 + timedelta(days=t2)
                elif isinstance(t1, int) and isinstance(t2, str):
                t2 = datetime.strptime(t2, TIME_FORMAT)
                t1 = t2 + timedelta(days=t1)
                else:
                assert False, "invalid time"
                if t1 > t2:
                t1, t2 = t2, t1
                query = f"{query} after:{t1.strftime(TIME_FORMAT)}
before:{t2.strftime(TIME_FORMAT)}"
```

```
return query.strip()

def sort_by_time(data: List[Dict]):
if order_by_time == "descend":
        reverse = True
else:
        reverse = False
sorted_data = sorted(
        data,
        key=lambda x: datetime.strptime(x["time"], "%Y-%m-%d %H:%M:%S"),
        reverse=reverse,
)
return sorted_data

try:
service = build("gmail", "v1", credentials=credential)
results = (
        service.users()
        .messages()
        .list(userId="me", labelIds=["INBOX"], q=generate_query())
        .execute()
)

messages = results.get("messages", [])
email_data = list()

if not messages:
        print("No eligible emails.")
        return None
else:
        pbar = tqdm(total=min(MAX_SEARCH_CNT, len(messages)))
        for cnt, message in enumerate(messages):
        pbar.update(1)
        if cnt >= MAX_SEARCH_CNT:
        break
        msg = (
        service.users()
        .messages()
        .get(
                userId="me",
                id=message["id"],
                format="full",
                metadataHeaders=None,
        )
        .execute()
        )

        subject = ""
```

```
            for header in msg["payload"]["headers"]:
            if header["name"] == "Subject":
                    subject = header["value"]
                    break

            sender = ""
            for header in msg["payload"]["headers"]:
            if header["name"] == "From":
                    sender = re.findall(
                    r"\b[\w\.-]+@[\w\.-]+\.\w+\b", header["value"]
                    )[0]
                    break
            body = ""
            if "parts" in msg["payload"]:
            for part in msg["payload"]["parts"]:
                    if part["mimeType"] == "text/plain":
                    data = part["body"]["data"]
                    body = base64.urlsafe_b64decode(data).decode("utf-8")
                    break

            email_info = {
            "sender": sender,
            "time": datetime.fromtimestamp(
                    int(msg["internalDate"]) / 1000
            ).strftime("%Y-%m-%d %H:%M:%S"),
            "subject": subject,
            "body": body,
            }
            email_data.append(email_info)
            pbar.close()
    email_data = sort_by_time(email_data)[0:number]
    return {"results": email_data}
    except Exception as e:
    print(e)
    return None

    def _send(self, mail_dict: dict):
    recipient_mail = mail_dict["recipient_mail"]
    subject = mail_dict["subject"]
    body = mail_dict["body"]
    credential = self.credential
    service = build("gmail", "v1", credentials=credential)

    message = MIMEMultipart()
    message["to"] = recipient_mail
    message["subject"] = subject

    message.attach(MIMEText(body, "plain"))
    raw_message = base64.urlsafe_b64encode(message.as_bytes()).decode("utf-8")
    try:
    message = (
        service.users()
        .messages()
        .send(userId="me", body={"raw": raw_message})
        .execute()
    )
    return {"state": True}
    except HttpError as error:
    print(error)
    return {"state": False}

    def convert_action_to(self, action_name: str):
    assert (
    action_name.lower() in self.__VALID_ACTION__
    ), f"Action `{action_name}` is not allowed! The valid action is in
`{self.__VALID_ACTION__}`"
    self.action = action_name.lower()

    def func(self, mail_dict: dict):
    if "action" in mail_dict:
    assert mail_dict["action"].lower() in self.__VALID_ACTION__
    self.action = mail_dict["action"]
    functions = {"read": self._read, "send": self._send}
    return functions[self.action](mail_dict)
```

tool.py

```
from abc import abstractmethod


class Tool:
    def __init__(self, description, name, parameters):
    self.type = "function"
    self.description = description  # A description of what the function does, used by the
model to choose when and how to call the function.
    self.name = name  # The name of the function to be called. Must be a-z, A-Z, 0-9, or
contain underscores and dashes, with a maximum length of 64.
    self.parameters = parameters  # The parameters the functions accepts, described as
a JSON Schema object. See the guide for examples, and the JSON Schema reference for
documentation about the format. Omitting parameters defines a function with an empty
parameter list.

    @abstractmethod
    def func(self):
    pass
```

```
        @abstractmethod
        def to_dict(self):
        pass

        @abstractmethod
        def load_from_dict(cls, dict_data):
        Pass
```

weather.py

```
import requests
from typing import Dict
from datetime import datetime, timedelta

from .tool import Tool


class WeatherTool(Tool):
        def __init__(self, api_key, TIME_FORMAT="%Y-%m-%d"):
        description = "Get historical weather data"
        name = "weather"
        parameters = {
        "city_name": {
                "type": "string",
                "description": "The name of the city",
        },
        "country_code": {
                "type": "string",
                "description": "The country code of the city",
        },
        "start_date": {
                "type": "string",
                "description": "The start date of the weather data",
        },
        "end_date": {
                "type": "string",
                "description": "The end date of the weather data",
        },
        }
        super(WeatherTool, self).__init__(description, name, parameters)
        self.TIME_FORMAT = TIME_FORMAT
        self.api_key = api_key

        def _parse(self, data):
        dict_data: dict = {}
        for item in data["data"]:
        date = item["datetime"]
        dict_data[date] = {}
        if "weather" in item:
                dict_data[date]["description"] = item["weather"]["description"]
        mapping = {
                "temp": "temperature",
                "max_temp": "max_temperature",
                "min_temp": "min_temperature",
                "precip": "accumulated_precipitation",
        }
        for key in ["temp", "max_temp", "min_temp", "precip"]:
                if key in item:
                dict_data[date][mapping[key]] = item[key]
        return dict_data

        def _query(self, city_name, country_code, start_date, end_date):
        """https://www.weatherbit.io/api/historical-weather-daily"""
        # print(datetime.strftime(start_date, self.TIME_FORMAT),
datetime.strftime(datetime.now(), self.TIME_FORMAT), end_date,
datetime.strftime(datetime.now()+timedelta(days=1), self.TIME_FORMAT))
        if start_date == datetime.strftime(
        datetime.now(), self.TIME_FORMAT
        ) and end_date == datetime.strftime(
        datetime.now() + timedelta(days=1), self.TIME_FORMAT
        ):
        """today"""
        url =
f"https://api.weatherbit.io/v2.0/current?city={city_name}&country={country_code}&key={self.
api_key}"
        else:
        url =
f"https://api.weatherbit.io/v2.0/history/daily?&city={city_name}&country={country_code}&start
_date={start_date}&end_date={end_date}&key={self.api_key}"
        response = requests.get(url)
        data = response.json()
        return self._parse(data)

        def func(self, weather_dict: Dict) -> Dict:
        TIME_FORMAT = self.TIME_FORMAT
        # Beijing, Shanghai
        city_name = weather_dict["city_name"]
        # CN, US
        country_code = weather_dict["country_code"]
        # 2020-02-02
        start_date = datetime.strftime(
        datetime.strptime(weather_dict["start_date"], self.TIME_FORMAT),
        self.TIME_FORMAT,
        )
        end_date = weather_dict["end_date"] if "end_date" in weather_dict else None
        if end_date is None:
```

```
        end_date = datetime.strftime(
                datetime.strptime(start_date, TIME_FORMAT) + timedelta(days=-1),
                TIME_FORMAT,
        )
        else:
        end_date = datetime.strftime(
                datetime.strptime(weather_dict["end_date"], self.TIME_FORMAT),
                self.TIME_FORMAT,
        )
        if datetime.strptime(start_date, TIME_FORMAT) > datetime.strptime(
        end_date, TIME_FORMAT
        ):
        start_date, end_date = end_date, start_date
        assert start_date != end_date
        return self._query(city_name, country_code, start_date, end_date)
```

Hawkindb as memory

```
import os
import json
import logging
from typing import Dict, Any, Optional, List, Union
from openai import OpenAI
from hawkinsdb import HawkinsDB

os.environ["OPENAI_API_KEY"]=""

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class TextToHawkinsDB:
        def __init__(self, api_key: Optional[str] = None):
        """Initialize with OpenAI API key."""
        self.api_key = api_key or os.getenv("OPENAI_API_KEY")
        if not self.api_key:
        raise ValueError("OpenAI API key is required")
        self.client = OpenAI(api_key=self.api_key)
        self.db = HawkinsDB(storage_type='sqlite')

        def text_to_json(self, text: str) -> Dict[str, Any]:
        """Convert text description to HawkinsDB-compatible JSON using GPT-4."""
        prompt = """Convert the following text into a structured JSON format suitable for a
memory database.

        Rules:
        1. Extract key entity details, properties, and relationships
        2. Use underscores for entity names (e.g., Python_Language)
        3. Categorize memory as one of: Semantic, Episodic, or Procedural
```

```
4. Include relevant properties and relationships

Required JSON format:
{
"column": "memory_type",
"name": "entity_name",
"properties": {
        "key1": "value1",
        "key2": ["value2a", "value2b"]
},
"relationships": {
        "related_to": ["entity1", "entity2"],
        "part_of": ["parent_entity"]
}
}

Text to convert:
"""

try:
response = self.client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[
        {"role": "system", "content": prompt},
        {"role": "user", "content": text}
        ],
        temperature=0.3,
        response_format={"type": "json_object"}
)

json_str = response.choices[0].message.content
return json.loads(json_str)

except Exception as e:
logger.error(f"Error converting text to JSON: {str(e)}")
raise

def add_to_db(self, text: str) -> Dict[str, Any]:
"""Convert text to JSON and add to HawkinsDB."""
try:
json_data = self.text_to_json(text)
logger.info(f"Converted JSON: {json.dumps(json_data, indent=2)}")

result = self.db.add_entity(json_data)
return {
        "success": True,
        "message": "Successfully added to database",
        "entity_data": json_data,
        "db_result": result
}

except Exception as e:
logger.error(f"Error adding to database: {str(e)}")
return {
        "success": False,
        "message": str(e),
        "entity_data": None,
        "db_result": None
}

def query_entity(self, entity_name: str) -> Dict[str, Any]:
"""Query specific entity by name."""
try:
frames = self.db.query_frames(entity_name)
if not frames:
        return {
        "success": False,
        "message": f"No entity found with name: {entity_name}",
        "data": None
        }

return {
        "success": True,
        "message": "Entity found",
        "data": frames
}

except Exception as e:
logger.error(f"Error querying entity: {str(e)}")
return {
        "success": False,
        "message": str(e),
        "data": None
}

def query_by_text(self, query_text: str) -> Dict[str, Any]:
"""Query database using natural language text."""
try:
# Get all entities for context
entities = self.db.list_entities()
if not entities:
        return {
        "success": True,
        "message": "Database is empty",
        "response": "No information available in the database."
        }
```

```
# Build context from existing entities
context = []
for entity_name in entities[:5]:  # Limit to 5 most recent entities
        frames = self.db.query_frames(entity_name)
        if frames:
        context.append(json.dumps(frames, indent=2))

# Create prompt with context
prompt = f"""You are a helpful assistant with access to a knowledge base.
Answer the following question based on this context:

Context:
{' '.join(context)}

Question: {query_text}

Rules:
1. Only use information from the provided context
2. If information is not in the context, say so
3. Be specific and include details when available
4. Format numbers and dates clearly
"""

# Get response from GPT-4
response = self.client.chat.completions.create(
        model="gpt4o",
        messages=[
        {"role": "system", "content": prompt}
        ],
        temperature=0.3,
        max_tokens=500
)

answer = response.choices[0].message.content

return {
        "success": True,
        "message": "Query processed successfully",
        "response": answer
}

except Exception as e:
logger.error(f"Error processing query: {str(e)}")
return {
        "success": False,
        "message": str(e),
        "response": None
        }

        def list_all_entities(self) -> Dict[str, Any]:
        """List all entities in the database."""
        try:
        entities = self.db.list_entities()
        return {
                "success": True,
                "message": "Entities retrieved successfully",
                "entities": entities
        }
        except Exception as e:
        logger.error(f"Error listing entities: {str(e)}")
        return {
                "success": False,
                "message": str(e),
                "entities": None
        }

def test_memory_examples():
        """Test function to demonstrate usage."""
        converter = TextToHawkinsDB()

        # Test adding entries
        examples = [
        """
        Python is a programming language created by Guido van Rossum in 1991.
        It supports object-oriented, imperative, and functional programming.
        It's commonly used for web development, data science, and automation.
        """,
        """
        Today I completed my first Python project in my home office.
        It took 2 hours and was successful. I did a code review afterwards.
        """,
        """
        The Tesla Model 3 is red, made in 2023, and parked in the garage.
        It has a range of 358 miles and goes 0-60 mph in 3.1 seconds.
        """
        ]

        # Add examples to database
        logger.info("\nAdding examples to database:")
        for i, example in enumerate(examples, 1):
        logger.info(f"\nAdding Example {i}")
        logger.info("=" * 50)
        result = converter.add_to_db(example)
        logger.info(f"Result: {json.dumps(result, indent=2)}")
        # Test queries
        logger.info("\nTesting queries:")

        # List all entities
        logger.info("\nListing all entities:")
        entities_result = converter.list_all_entities()
        logger.info(f"Entities: {json.dumps(entities_result, indent=2)}")

        # Query specific entity
        logger.info("\nQuerying specific entity:")
        entity_result = converter.query_entity("Python_Language")
        print(entity_result)

        # Test natural language queries
        test_queries = [
        "What programming language was created by Guido van Rossum?",
        "Tell me about the Tesla Model 3's specifications.",
        "What happened during the first Python project?"
        ]

        logger.info("\nTesting natural language queries:")
        for query in test_queries:
        logger.info(f"\nQuery: {query}")
        result = converter.query_by_text(query)
        logger.info(f"Response: {json.dumps(result, indent=2)}")

if __name__ == "__main__":
        test_memory_examples()
```