## agent_management.py

```
import base64
import os
import re
import streamlit as st

from config import API_URL, MODEL_CHOICES, MODEL_TOKEN_LIMITS

from ui_utils import get_llm_provider, regenerate_json_files_and_zip, update_discussion_and_whiteboard


def agent_button_callback(agent_index):
# Callback function to handle state update and logic execution
def callback():
st.session_state['selected_agent_index'] = agent_index
agent = st.session_state.agents[agent_index]
agent_name = agent['config']['name'] if 'config' in agent and 'name' in agent['config'] else ''
st.session_state['form_agent_name'] = agent_name
st.session_state['form_agent_description'] = agent['description'] if 'description' in agent else ''
# Directly call process_agent_interaction here if appropriate
process_agent_interaction(agent_index)
return callback


def construct_request(agent_name, description, user_request, user_input, rephrased_request, reference_url,
skill_results):
request = f"Act as the {agent_name} who {description}."
if user_request:
request += f" Original request was: {user_request}."
if rephrased_request:
request += f" You are helping a team work on satisfying {rephrased_request}."
if user_input:
request += f" Additional input: {user_input}."
if reference_url and reference_url in st.session_state.reference_html:
html_content = st.session_state.reference_html[reference_url]
request += f" Reference URL content: {html_content}."
if st.session_state.discussion:
request += f" The discussion so far has been {st.session_state.discussion[-50000:]}."
if skill_results:
request += f" Skill results: {skill_results}."
return request


def display_agents():
if "agents" in st.session_state and st.session_state.agents:
st.sidebar.title("Your Agents")
st.sidebar.subheader("Click to interact")
display_agent_buttons(st.session_state.agents)
if st.session_state.get('show_edit'):
```

```
edit_index = st.session_state.get('edit_agent_index')
if edit_index is not None and 0 <= edit_index < len(st.session_state.agents):
agent = st.session_state.agents[edit_index]
display_agent_edit_form(agent, edit_index)
else:
st.sidebar.warning("Invalid agent selected for editing.")
else:
st.sidebar.warning(f"No agents have yet been created. Please enter a new request.")
st.sidebar.warning(f"NOTE: GPT models can only be used locally, not in the online demo.")
st.sidebar.warning(f"ALSO: If no agents are created, do a hard reset (CTL-F5) and try switching models. LLM results can be unpredictable.")
st.sidebar.warning(f"SOURCE:  https://github.com/jgravelle/AutoGroq\n\r\n\r https://j.gravelle.us")


def display_agent_buttons(agents):
for index, agent in enumerate(agents):
agent_name = agent["config"]["name"] if agent["config"].get("name") else f"Unnamed Agent {index + 1}"
col1, col2 = st.sidebar.columns([1, 4])
with col1:
gear_icon = "" # Unicode character for gear icon
if st.button(
gear_icon,
key=f"gear_{index}",
help="Edit Agent" # Add the tooltip text
):
st.session_state['edit_agent_index'] = index
st.session_state['show_edit'] = True
with col2:
if "next_agent" in st.session_state and st.session_state.next_agent == agent_name:
button_style = """
<style>
div[data-testid*="stButton"] > button[kind="secondary"] {
background-color: green !important;
color: white !important;
}
</style>
"""
st.markdown(button_style, unsafe_allow_html=True)
st.button(agent_name, key=f"agent_{index}", on_click=agent_button_callback(index))


def display_agent_edit_form(agent, edit_index):
with st.expander(f"Edit Properties of {agent['config'].get('name', '')}", expanded=True):
col1, col2 = st.columns([4, 1])
with col1:
new_name = st.text_input("Name", value=agent['config'].get('name', ''), key=f"name_{edit_index}")
with col2:
container = st.container()
if container.button("X", key=f"delete_{edit_index}"):
if st.session_state.get(f"delete_confirmed_{edit_index}", False):
```

```
st.session_state.agents.pop(edit_index)
st.session_state['show_edit'] = False
st.experimental_rerun()
else:
st.session_state[f"delete_confirmed_{edit_index}"] = True
st.experimental_rerun()
if st.session_state.get(f"delete_confirmed_{edit_index}", False):
if container.button("Confirm Deletion", key=f"confirm_delete_{edit_index}"):
st.session_state.agents.pop(edit_index)
st.session_state['show_edit'] = False
del st.session_state[f"delete_confirmed_{edit_index}"]
st.experimental_rerun()
if container.button("Cancel", key=f"cancel_delete_{edit_index}"):
del st.session_state[f"delete_confirmed_{edit_index}"]
st.experimental_rerun()
description_value = agent.get('new_description', agent.get('description', ''))

col1, col2 = st.columns([3, 1])
with col1:
selected_model = st.selectbox("Model", options=list(MODEL_CHOICES.keys()),
index=list(MODEL_CHOICES.keys()).index(agent['config']['llm_config']['config_list'][0]['model']),
key=f"model_select_{edit_index}")
with col2:
if st.button("Set for ALL agents", key=f"set_all_agents_{edit_index}"):
for agent in st.session_state.agents:
agent['config']['llm_config']['config_list'][0]['model'] = selected_model
agent['config']['llm_config']['max_tokens'] = MODEL_CHOICES[selected_model]
st.experimental_rerun()

new_description = st.text_area("Description", value=description_value, key=f"desc_{edit_index}")

col1, col2 = st.columns([3, 1])
with col1:
if st.button("Update User Description", key=f"regenerate_{edit_index}"):
print(f"Regenerate button clicked for agent {edit_index}")
new_description = regenerate_agent_description(agent)
if new_description:
agent['new_description'] = new_description
print(f"Description regenerated for {agent['config']['name']}: {new_description}")
st.session_state[f"regenerate_description_{edit_index}"] = True
description_value = new_description
st.experimental_rerun()
else:
print(f"Failed to regenerate description for {agent['config']['name']}")
with col2:
if st.button("Save Changes", key=f"save_{edit_index}"):
agent['config']['name'] = new_name
agent['description'] = agent.get('new_description', new_description)

if selected_model != 'default':
```

```
agent['config']['llm_config']['config_list'][0]['model'] = selected_model
agent['config']['llm_config']['max_tokens'] = MODEL_CHOICES[selected_model]
else:
agent['config']['llm_config']['config_list'][0]['model'] = st.session_state.model
agent['config']['llm_config']['max_tokens'] = MODEL_TOKEN_LIMITS.get(st.session_state.model, 4096)

st.session_state['show_edit'] = False
if 'edit_agent_index' in st.session_state:
del st.session_state['edit_agent_index']
if 'new_description' in agent:
del agent['new_description']
st.session_state.agents[edit_index] = agent
regenerate_json_files_and_zip()
st.session_state['show_edit'] = False


def download_agent_file(expert_name):
# Format the expert_name
formatted_expert_name = re.sub(r'[^a-zA-Z0-9\s]', '', expert_name) # Remove non-alphanumeric characters
formatted_expert_name = formatted_expert_name.lower().replace(' ', '_') # Convert to lowercase and replace spaces
with underscores
# Get the full path to the agent JSON file
agents_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), "agents"))
json_file = os.path.join(agents_dir, f"{formatted_expert_name}.json")
# Check if the file exists
if os.path.exists(json_file):
# Read the file content
with open(json_file, "r") as f:
file_content = f.read()
# Encode the file content as base64
b64_content = base64.b64encode(file_content.encode()).decode()
# Create a download link
href = f'<a href="data:application/json;base64,{b64_content}" download="{formatted_expert_name}.json">Download
{formatted_expert_name}.json</a>'
st.markdown(href, unsafe_allow_html=True)
else:
st.error(f"File not found: {json_file}")


def process_agent_interaction(agent_index):
agent_name, description = retrieve_agent_information(agent_index)
user_request = st.session_state.get('user_request', '')
user_input = st.session_state.get('user_input', '')
rephrased_request = st.session_state.get('rephrased_request', '')
reference_url = st.session_state.get('reference_url', '')
# Execute associated skills for the agent
agent = st.session_state.agents[agent_index]
agent_skills = agent.get("skills", [])
skill_results = {}
for skill_name in agent_skills:
```

```
if skill_name in st.session_state.skill_functions:
skill_function = st.session_state.skill_functions[skill_name]
skill_result = skill_function()
skill_results[skill_name] = skill_result
request = construct_request(agent_name, description, user_request, user_input, rephrased_request, reference_url, skill_results)
print(f"Request: {request}")
# Use the dynamic LLM provider to send the request
llm_provider = get_llm_provider(API_URL)
llm_request_data = {
"model": st.session_state.model,
"temperature": st.session_state.get('temperature', 0.1),
"max_tokens": st.session_state.max_tokens,
"top_p": 1,
"stop": "TERMINATE",
"messages": [
{
"role": "user",
"content": request
}
]
}
response = llm_provider.send_request(llm_request_data)
if response.status_code == 200:
response_data = llm_provider.process_response(response)
if "choices" in response_data and response_data["choices"]:
content = response_data["choices"][0]["message"]["content"]
update_discussion_and_whiteboard(agent_name, content, user_input)
st.session_state['form_agent_name'] = agent_name
st.session_state['form_agent_description'] = description
st.session_state['selected_agent_index'] = agent_index


def regenerate_agent_description(agent):
agent_name = agent['config']['name']
print(f"agent_name: {agent_name}")
agent_description = agent['description']
print(f"agent_description: {agent_description}")
user_request = st.session_state.get('user_request', '')
print(f"user_request: {user_request}")
discussion_history = st.session_state.get('discussion_history', '')
prompt = f"""
You are an AI assistant helping to improve an agent's description. The agent's current details are:
Name: {agent_name}
Description: {agent_description}
The current user request is: {user_request}
The discussion history so far is: {discussion_history}
Please generate a revised description for this agent that defines it in the best manner possible to address the current user request, taking into account the discussion thus far. Return only the revised description, written in the third-person, without any additional commentary or narrative. It is imperative that you return ONLY the text of the new
```

description written in the third-person. No preamble, no narrative, no superfluous commentary whatsoever. Just the description, written in the third-person, unlabeled, please.  You will have been successful if your reply is thorough, comprehensive, concise, written in the third-person, and adherent to all of these instructions.
"""
print(f"regenerate_agent_description called with agent_name: {agent_name}")
print(f"regenerate_agent_description called with prompt: {prompt}")
llm_provider = get_llm_provider(API_URL)
llm_request_data = {
"model": st.session_state.model,
"temperature": st.session_state.get('temperature', 0.1),
"max_tokens": st.session_state.max_tokens,
"top_p": 1,
"stop": "TERMINATE",
"messages": [
{
"role": "user",
"content": prompt
}
]
}
response = llm_provider.send_request(llm_request_data)
if response.status_code == 200:
response_data = llm_provider.process_response(response)
if "choices" in response_data and response_data["choices"]:
content = response_data["choices"][0]["message"]["content"]
return content.strip()
return None


def retrieve_agent_information(agent_index):
agent = st.session_state.agents[agent_index]
agent_name = agent["config"]["name"]
description = agent["description"]
return agent_name, description


def send_request(agent_name, request):
llm_provider = get_llm_provider(API_URL)
response = llm_provider.send_request(request)
return response

## api_utils.py

```
import importlib
import requests
import streamlit as st
import time

from config import LLM_PROVIDER, RETRY_TOKEN_LIMIT
```

```
def get_llm_provider(api_url):
provider_module = importlib.import_module(f"llm_providers.{LLM_PROVIDER}_provider")
provider_class = getattr(provider_module, f"{LLM_PROVIDER.capitalize()}Provider")
return provider_class(api_url=api_url)


def make_api_request(url, data, headers, api_key):
time.sleep(2)  # Throttle the request to ensure at least 2 seconds between calls
try:
if not api_key:
llm = LLM_PROVIDER.upper()
raise ValueError(f"{llm}_API_KEY not found. Please enter your API key.")
headers["Authorization"] = f"Bearer {api_key}"
response = requests.post(url, json=data, headers=headers)
if response.status_code == 200:
return response.json()
elif response.status_code == 429:
error_message = response.json().get("error", {}).get("message", "")
st.error(f"Rate limit reached for the current model. If you click 'Update' again, we'll retry with a reduced token count.
Or you can try selecting a different model.")
st.error(f"Error details: {error_message}")
return None
else:
print(f"Error: API request failed with status {response.status_code}, response: {response.text}")
return None
except requests.RequestException as e:
print(f"Error: Request failed {e}")
return None


def send_request_with_retry(url, data, headers, api_key):
response = make_api_request(url, data, headers, api_key)
if response is None:
# Add a retry button
if st.button("Retry with decreased token limit"):
# Update the token limit in the request data
data["max_tokens"] = RETRY_TOKEN_LIMIT
# Retry the request with the decreased token limit
print(f"Retrying the request with decreased token limit.")
print(f"URL: {url}")
print(f"Retry token limit: {RETRY_TOKEN_LIMIT}")
response = make_api_request(url, data, headers, api_key)
if response is not None:
print(f"Retry successful. Response: {response}")
else:
print("Retry failed.")
return response
```

## auth_utils.py

```
import os
import streamlit as st

from config import LLM_PROVIDER


def get_api_key():
api_key_env_var = f"{LLM_PROVIDER.upper()}_API_KEY"
api_key = os.environ.get(api_key_env_var)
if api_key is None:
api_key = globals().get(api_key_env_var)
if api_key is None:
if api_key_env_var not in st.session_state:
api_key = st.text_input(f"Enter the {LLM_PROVIDER.upper()} API Key:", type="password",
key=f"{LLM_PROVIDER}_api_key_input")
if api_key:
st.session_state[api_key_env_var] = api_key
st.success("API Key entered successfully.")
else:
st.warning(f"Please enter the {LLM_PROVIDER.upper()} API Key to use the app.")
else:
api_key = st.session_state.get(api_key_env_var)
return api_key

def get_api_url():
api_url_env_var = f"{LLM_PROVIDER.upper()}_API_URL"
api_url = os.environ.get(api_url_env_var)
if api_url is None:
api_url = globals().get(api_url_env_var)
if api_url is None:
if api_url_env_var not in st.session_state:
api_url = st.text_input(f"Enter the {LLM_PROVIDER.upper()} API URL:", type="password",
key=f"{LLM_PROVIDER}_api_url_input")
if api_url:
st.session_state[api_url_env_var] = api_url
st.success("API URL entered successfully.")
else:
st.warning(f"Please enter the {LLM_PROVIDER.upper()} API URL to use the app.")
else:
api_url = st.session_state.get(api_url_env_var)
return api_url
```

## config.py

```
#APIs
LLM_PROVIDER = "groq" # Supported values: "groq", "openai", "ollama", "lmstudio"


GROQ_API_URL = "https://api.groq.com/openai/v1/chat/completions"
LMSTUDIO_API_URL = "http://localhost:1234/v1/chat/completions"
OLLAMA_API_URL = "http://127.0.0.1:11434/api/generate"
```

```
OPENAI_API_KEY = None
OPENAI_API_URL = "https://api.openai.com/v1/chat/completions"

if LLM_PROVIDER == "groq":
API_KEY_NAME = "GROQ_API_KEY"
API_URL = GROQ_API_URL
elif LLM_PROVIDER == "lmstudio":
API_KEY_NAME = None
API_URL = LMSTUDIO_API_URL
elif LLM_PROVIDER == "openai":
API_KEY_NAME = "OPENAI_API_KEY"
API_URL = OPENAI_API_URL
elif LLM_PROVIDER == "ollama":
API_KEY_NAME = None
API_URL = OLLAMA_API_URL
else:
raise ValueError(f"Unsupported LLM provider: {LLM_PROVIDER}")

API_KEY_NAMES = {
"groq": "GROQ_API_KEY",
"lmstudio": None,
"ollama": None,
"openai": "OPENAI_API_KEY",
# Add other LLM providers and their respective API key names here
}

# Retry settings
MAX_RETRIES = 3
RETRY_DELAY = 2  # in seconds
RETRY_TOKEN_LIMIT = 5000
LLM_URL = GROQ_API_URL

# Model configurations
if LLM_PROVIDER == "groq":
MODEL_TOKEN_LIMITS = {
'mixtral-8x7b-32768': 32768,
'llama3-70b-8192': 8192,
'llama3-8b-8192': 8192,
'gemma-7b-it': 8192,
}
elif LLM_PROVIDER == "lmstudio":
MODEL_TOKEN_LIMITS = {
'instructlab/granite-7b-lab-GGUF': 2048,
}
elif LLM_PROVIDER == "openai":
MODEL_TOKEN_LIMITS = {
'gpt-4o': 4096,
}
elif LLM_PROVIDER == "ollama":
```

```
MODEL_TOKEN_LIMITS = {
'llama3': 8192,
}
else:
MODEL_TOKEN_LIMITS = {}



# Database path
AUTOGEN_DB_PATH = "C:\\Users\\j\\.autogenstudio\\database.sqlite"

MODEL_CHOICES = {
'default': None,
'gemma-7b-it': 8192,
'gpt-4o': 4096,
'instructlab/granite-7b-lab-GGUF': 2048,
'llama3': 8192,
'llama3-70b-8192': 8192,
'llama3-8b-8192': 8192,
'mixtral-8x7b-32768': 32768
}
```

## current_project.py

```
class Current_Project:
def __init__(self):
self.re_engineered_prompt = ""
self.objectives = []
self.deliverables = []

def set_re_engineered_prompt(self, prompt):
self.re_engineered_prompt = prompt

def add_objective(self, objective):
self.objectives.append({"text": objective, "done": False})

def add_deliverable(self, deliverable):
self.deliverables.append({"text": deliverable, "done": False})

def mark_objective_done(self, index):
if 0 <= index < len(self.objectives):
self.objectives[index]["done"] = True

def mark_deliverable_done(self, index):
if 0 <= index < len(self.deliverables):
self.deliverables[index]["done"] = True

def mark_objective_undone(self, index):
if 0 <= index < len(self.objectives):
self.objectives[index]["done"] = False

def mark_deliverable_undone(self, index):
```

if 0 <= index < len(self.deliverables):
self.deliverables[index]["done"] = False

## db_utils.py

```
import datetime
import json
import os
import sqlite3
import streamlit as st
import uuid

from config import AUTOGEN_DB_PATH, MODEL_CHOICES, MODEL_TOKEN_LIMITS
from file_utils import create_agent_data, create_skill_data, sanitize_text
from ui_utils import get_workflow_from_agents



def export_to_autogen():
# Check if the app is running on Streamlit Sharing
url_params = st.query_params
if "streamlit.app" in url_params.get("url", ""):
st.warning("Exporting to Autogen is only possible with a locally running copy of AutoGroq™.")
return

db_path = AUTOGEN_DB_PATH
print(f"Database path: {db_path}")
if db_path:
export_data(db_path)
else:
st.warning("Please provide a valid database path in config.py.")



def export_data(db_path):
print(f"Exporting data to: {db_path}")

if db_path:
try:
conn = sqlite3.connect(db_path)
cursor = conn.cursor()
print("Connected to the database successfully.")

# Access agents from st.session_state
agents = st.session_state.agents
print(f"Number of agents: {len(agents)}")

# Keep track of inserted skills to avoid duplicates
inserted_skills = set()

for agent in agents:
agent_name = agent['config']['name']
formatted_agent_name = sanitize_text(agent_name).lower().replace(' ', '_')
```

```
autogen_agent_data, _ = create_agent_data(agent)

# Update the model and max_tokens in the autogen_agent_data
autogen_agent_data['config']['llm_config']['config_list'][0]['model'] = agent['config']['llm_config']['config_list'][0]['model']
autogen_agent_data['config']['llm_config']['max_tokens'] =
MODEL_CHOICES.get(agent['config']['llm_config']['config_list'][0]['model'],
MODEL_TOKEN_LIMITS.get(st.session_state.model, 4096))

agent_data = (
str(uuid.uuid4()), # Generate a unique ID for the agent
'default',
datetime.datetime.now().isoformat(),
json.dumps(autogen_agent_data['config']),
autogen_agent_data['type'],
json.dumps(autogen_agent_data['skills'])
)
cursor.execute("INSERT INTO agents (id, user_id, timestamp, config, type, skills) VALUES (?, ?, ?, ?, ?, ?)",
agent_data)
print(f"Inserted agent: {formatted_agent_name}")

script_dir = os.path.dirname(os.path.abspath(__file__))
skill_folder = os.path.join(script_dir, "skills")
skill_files = [f for f in os.listdir(skill_folder) if f.endswith(".py")]
for skill_name in st.session_state.selected_skills:
if skill_name not in inserted_skills:
skill_file_path = os.path.join(skill_folder, f"{skill_name}.py")
with open(skill_file_path, 'r') as file:
skill_data = file.read()
skill_json = create_skill_data(skill_data)
skill_data = (
str(uuid.uuid4()),  # Generate a unique ID for the skill
'default',  # Set the user ID to 'default'
datetime.datetime.now().isoformat(),
skill_data,
skill_json['title'],
skill_json['file_name']
)
cursor.execute("INSERT INTO skills (id, user_id, timestamp, content, title, file_name) VALUES (?, ?, ?, ?, ?, ?)",
skill_data)
print(f"Inserted skill: {skill_json['title']}")
inserted_skills.add(skill_name)  # Add the inserted skill to the set

# Access agents from st.session_state for workflow
workflow_data = get_workflow_from_agents(st.session_state.agents)[0]
workflow_data = (
str(uuid.uuid4()),  # Generate a unique ID for the workflow
'default',
datetime.datetime.now().isoformat(),
json.dumps(workflow_data['sender']),
json.dumps(workflow_data['receiver']),
```

```
workflow_data['type'],
workflow_data['name'],
workflow_data['description'],
workflow_data['summary_method']
)
cursor.execute("INSERT INTO workflows (id, user_id, timestamp, sender, receiver, type, name, description,
summary_method) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)", workflow_data)
print("Inserted workflow data.")

conn.commit()
print("Changes committed to the database.")

conn.close()
print("Database connection closed.")

st.success("Data exported to Autogen successfully!")
except sqlite3.Error as e:
st.error(f"Error exporting data to Autogen: {str(e)}")
print(f"Error exporting data to Autogen: {str(e)}")
```

## file_utils.py

```
import datetime
import os
import re
import streamlit as st



def create_agent_data(agent):
expert_name = agent['config']['name']
description = agent['description']
current_timestamp = datetime.datetime.now().isoformat()

formatted_expert_name = sanitize_text(expert_name)
formatted_expert_name = formatted_expert_name.lower().replace(' ', '_')

sanitized_description = sanitize_text(description)
temperature_value = st.session_state.get('temperature', 0.1)

autogen_agent_data = {
"type": "assistant",
"config": {
"name": formatted_expert_name,
"llm_config": {
"config_list": [
{
"user_id": "default",
"timestamp": current_timestamp,
"model": st.session_state.model,
"base_url": None,
"api_type": None,
```

```
"api_version": None,
"description": "OpenAI model configuration"
}
],
"temperature": temperature_value,
"cache_seed": None,
"timeout": None,
"max_tokens": None,
"extra_body": None
},
"human_input_mode": "NEVER",
"max_consecutive_auto_reply": 8,
"system_message": f"You are a helpful assistant that can act as {expert_name} who {sanitized_description}.",
"is_termination_msg": None,
"code_execution_config": None,
"default_auto_reply": "",
"description": description
},
"timestamp": current_timestamp,
"user_id": "default",
"skills": []
}

#script_dir = os.path.dirname(os.path.abspath(__file__))
skill_folder = os.path.join(os.path.dirname(os.path.abspath(__file__)), "skills")
skill_files = [f for f in os.listdir(skill_folder) if f.endswith(".py")]

for skill_file in skill_files:
skill_name = os.path.splitext(skill_file)[0]
if agent.get(skill_name, False):
skill_file_path = os.path.join(skill_folder, skill_file)
with open(skill_file_path, 'r') as file:
skill_data = file.read()
skill_json = create_skill_data(skill_data)
autogen_agent_data["skills"].append(skill_json)

crewai_agent_data = {
"name": expert_name,
"description": description,
"verbose": True,
"allow_delegation": True
}

return autogen_agent_data, crewai_agent_data


def create_skill_data(python_code):
# Extract the function name from the Python code
function_name_match = re.search(r"def\s+(\w+)\(", python_code)
if function_name_match:
```

```
function_name = function_name_match.group(1)
else:
function_name = "unnamed_function"

# Extract the skill description from the docstring
docstring_match = re.search(r"""(.*?)""", python_code, re.DOTALL)
if docstring_match:
skill_description = docstring_match.group(1).strip()
else:
skill_description = "No description available"

# Get the current timestamp
current_timestamp = datetime.datetime.now().isoformat()

# Create the skill data dictionary
skill_data = {
"title": function_name,
"content": python_code,
"file_name": f"{function_name}.json",
"description": skill_description,
"timestamp": current_timestamp,
"user_id": "default"
}

return skill_data


def create_workflow_data(workflow):
# Sanitize the workflow name
sanitized_workflow_name = sanitize_text(workflow["name"])
sanitized_workflow_name = sanitized_workflow_name.lower().replace(' ', '_')

return workflow


def sanitize_text(text):
# Remove non-ASCII characters
text = re.sub(r'[^\x00-\x7F]+', '', text)
# Remove non-alphanumeric characters except for standard punctuation
text = re.sub(r'[^a-zA-Z0-9\s.,!?:;\'"-]+', '', text)
return text
```

## main.py

```
import os
import streamlit as st

from config import LLM_PROVIDER, MODEL_TOKEN_LIMITS

from agent_management import display_agents
from auth_utils import get_api_key
```

```
from db_utils import export_to_autogen
from ui_utils import display_api_key_input, display_discussion_and_whiteboard, display_download_button,
display_user_input, display_reset_and_upload_buttons, display_user_request_input, handle_user_request,
load_skill_functions


def main():
# Construct the relative path to the CSS file
css_file = "AutoGroq/style.css"

# Check if the CSS file exists
if os.path.exists(css_file):
with open(css_file) as f:
st.markdown(f'<style>{f.read()}</style>', unsafe_allow_html=True)
else:
st.error(f"CSS file not found: {os.path.abspath(css_file)}")

load_skill_functions()

api_key = get_api_key()
if api_key is None:
api_key = display_api_key_input()
if api_key is None:
llm = LLM_PROVIDER.upper()
st.warning(f"{llm}_API_KEY not found. Please enter your API key.")
return


col1, col2 = st.columns([1, 1])  # Adjust the column widths as needed
with col1:
selected_model = st.selectbox(
'Select Model',
options=list(MODEL_TOKEN_LIMITS.keys()),
index=0,
key='model_selection'
)
st.session_state.model = selected_model
st.session_state.max_tokens = MODEL_TOKEN_LIMITS[selected_model]

with col2:
temperature = st.slider(
"Set Temperature",
min_value=0.0,
max_value=1.0,
value=st.session_state.get('temperature', 0.3),
step=0.01,
key='temperature'
)

# If the LLM Provider is "groq", the title is "AutoGroq"
```

```
if LLM_PROVIDER == "groq":
st.title("AutoGroq")
elif LLM_PROVIDER == "ollama":
st.title("AutoGroqOllama")
elif LLM_PROVIDER == "lmstudio":
st.title("AutoGroqLM_Studio")
elif LLM_PROVIDER == "openai":
st.title("AutoGroqChatGPT")



# Ensure default values for session state are set
if "whiteboard" not in st.session_state:
st.session_state.whiteboard = "" # Apply CSS classes to elements

with st.sidebar:
st.markdown('<div class="sidebar">', unsafe_allow_html=True)
st.markdown('</div>', unsafe_allow_html=True)

display_agents()

with st.container():
st.markdown('<div class="main">', unsafe_allow_html=True)
if st.session_state.get("rephrased_request", "") == "":
user_request = st.text_input("Enter your request:", key="user_request", value=st.session_state.get("user_request", ""),
on_change=handle_user_request, args=(st.session_state,))
display_user_request_input()
# display_rephrased_request()
st.markdown('<div class="discussion-whiteboard">', unsafe_allow_html=True)
display_discussion_and_whiteboard()
st.markdown('</div>', unsafe_allow_html=True)
st.markdown('<div class="user-input">', unsafe_allow_html=True)
display_user_input()
st.markdown('</div>', unsafe_allow_html=True)
display_reset_and_upload_buttons()
st.markdown('</div>', unsafe_allow_html=True)

if "autogen_zip_buffer" in st.session_state and "crewai_zip_buffer" in st.session_state:
display_download_button()
if st.button("Export to Autogen"):
export_to_autogen()



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

## ui_utils.py

```
import datetime
import importlib
import os
import streamlit as st
```

```
import time

from config import API_URL, LLM_PROVIDER, MAX_RETRIES, MODEL_TOKEN_LIMITS, RETRY_DELAY
from current_project import Current_Project

from skills.fetch_web_content import fetch_web_content



def display_api_key_input():
if 'api_key' not in st.session_state:
st.session_state.api_key = ''
llm = LLM_PROVIDER.upper()
api_key = st.text_input(f"Enter your {llm}_API_KEY:", type="password", value=st.session_state.api_key,
key="api_key_input")

if api_key:
st.session_state.api_key = api_key
st.success("API key entered successfully.")
print(f"API Key: {api_key}")

return api_key



import io
import json
import pandas as pd
import re
import time
import zipfile

from api_utils import get_llm_provider
from file_utils import create_agent_data, create_skill_data, sanitize_text

import datetime

def create_project_manager(rephrased_text, api_url):
temperature_value = st.session_state.get('temperature', 0.1)
llm_request_data = {
"model": st.session_state.model,
"temperature": temperature_value,
"max_tokens": st.session_state.max_tokens,
"top_p": 1,
"stop": "TERMINATE",
"messages": [
{
"role": "user",
"content": f"""
You are a Project Manager tasked with creating a comprehensive project outline and describing the perfect team of
experts that should be created to work on the following project:
```

{rephrased_text}

Please provide a detailed project outline, including the objectives, key deliverables, and timeline. Also, describe the ideal team of experts required for this project, including their roles, skills, and responsibilities.  Your analysis shall consider the complexity, domain, and specific needs of the request to assemble a multidisciplinary team of experts. The team should be as small as possible while still providing a complete and comprehensive talent pool able to properly address the user's request. Each recommended agent shall come with a defined role, a brief but thorough description of their expertise, their specific skills, and the specific tools they would utilize to achieve the user's goal.

Return your response in the following format:

Project Outline:
[Detailed project outline]

Team of Experts:
[Description of the ideal team of experts]
"""
}
]
}

llm_provider = get_llm_provider(api_url)
response = llm_provider.send_request(llm_request_data)

if response.status_code == 200:
response_data = llm_provider.process_response(response)
if "choices" in response_data and response_data["choices"]:
content = response_data["choices"][0]["message"]["content"]
return content.strip()

return None


def create_zip_file(zip_buffer, file_data):
with zipfile.ZipFile(zip_buffer, 'w', zipfile.ZIP_DEFLATED) as zip_file:
for file_name, file_content in file_data.items():
zip_file.writestr(file_name, file_content)


def display_discussion_and_whiteboard():
discussion_history = get_discussion_history()

tab1, tab2, tab3, tab4, tab5, tab6, tab7 = st.tabs(["Most Recent Comment", "Whiteboard", "Discussion History", "Objectives", "Deliverables", "Goal", "Skills"])

with tab1:
if "last_comment" not in st.session_state:
st.session_state.last_comment = ""
st.text_area("Most Recent Comment", value=st.session_state.last_comment, height=400, key="discussion")

```
with tab2:
if "whiteboard" not in st.session_state:
st.session_state.whiteboard = ""
st.text_area("Whiteboard", value=st.session_state.whiteboard, height=400, key="whiteboard")

with tab3:
st.write(discussion_history)

with tab4:
if "current_project" in st.session_state:
current_project = st.session_state.current_project
for index, objective in enumerate(current_project.objectives):
if objective["text"].strip():  # Check if the objective text is not empty
checkbox_key = f"objective_{index}"
done = st.checkbox(objective["text"], value=objective["done"], key=checkbox_key)
if done != objective["done"]:
if done:
current_project.mark_objective_done(index)
else:
current_project.mark_objective_undone(index)
else:
st.warning("No objectives found. Please enter a user request.")

with tab5:
if "current_project" in st.session_state:
current_project = st.session_state.current_project
for index, deliverable in enumerate(current_project.deliverables):
if deliverable["text"].strip():  # Check if the deliverable text is not empty
checkbox_key = f"deliverable_{index}"
done = st.checkbox(deliverable["text"], value=deliverable["done"], key=checkbox_key)
if done != deliverable["done"]:
if done:
current_project.mark_deliverable_done(index)
else:
current_project.mark_deliverable_undone(index)

with tab6:
rephrased_request = st.text_area("Re-engineered Prompt:", value=st.session_state.get('rephrased_request', ''),
height=100, key="rephrased_request_area")

with tab7:
script_dir = os.path.dirname(os.path.abspath(__file__))
skill_folder = os.path.join(script_dir, "skills")
skill_files = [f for f in os.listdir(skill_folder) if f.endswith(".py")]
selected_skills = []

select_all = st.checkbox("Select All", key="select_all_skills")

for skill_file in skill_files:
skill_name = os.path.splitext(skill_file)[0]
```

```
if select_all:
skill_checkbox = st.checkbox(f"Add {skill_name} skill to all agents", value=True, key=f"skill_{skill_name}")
else:
skill_checkbox = st.checkbox(f"Add {skill_name} skill to all agents", value=False, key=f"skill_{skill_name}")
if skill_checkbox:
selected_skills.append(skill_name)

if select_all:
st.session_state.selected_skills = [os.path.splitext(f)[0] for f in skill_files]
else:
st.session_state.selected_skills = selected_skills

regenerate_zip_files()


def display_download_button():
col1, col2 = st.columns(2)
with col1:
st.download_button(
label="Download Autogen Files",
data=st.session_state.autogen_zip_buffer,
file_name="autogen_files.zip",
mime="application/zip",
key=f"autogen_download_button_{int(time.time())}"  # Generate a unique key based on timestamp
)
with col2:
st.download_button(
label="Download CrewAI Files",
data=st.session_state.crewai_zip_buffer,
file_name="crewai_files.zip",
mime="application/zip",
key=f"crewai_download_button_{int(time.time())}"  # Generate a unique key based on timestamp
)


def display_user_input():
user_input = st.text_area("Additional Input:", key="user_input", height=100)
reference_url = st.text_input("URL:", key="reference_url")

if user_input:
url_pattern = re.compile(r'http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\\(\\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+')
url_match = url_pattern.search(user_input)
if url_match:
url = url_match.group()
if "reference_html" not in st.session_state or url not in st.session_state.reference_html:
html_content = fetch_web_content(url)
if html_content:
if "reference_html" not in st.session_state:
st.session_state.reference_html = {}
st.session_state.reference_html[url] = html_content
```

```
else:
st.warning("Failed to fetch HTML content.")
else:
st.session_state.reference_html = {}
else:
st.session_state.reference_html = {}
else:
st.session_state.reference_html = {}

return user_input, reference_url


def display_reset_and_upload_buttons():
col1, col2 = st.columns(2)
with col1:
if st.button("Reset", key="reset_button"):
# Define the keys of session state variables to clear
keys_to_reset = [
"rephrased_request", "discussion", "whiteboard", "user_request",
"user_input", "agents", "zip_buffer", "crewai_zip_buffer",
"autogen_zip_buffer", "uploaded_file_content", "discussion_history",
"last_comment", "user_api_key", "reference_url"
]
# Reset each specified key
for key in keys_to_reset:
if key in st.session_state:
del st.session_state[key]
# Additionally, explicitly reset user_input to an empty string
st.session_state.user_input = ""
st.session_state.show_begin_button = True
st.experimental_rerun()

with col2:
uploaded_file = st.file_uploader("Upload a sample .csv of your data (optional)", type="csv")

if uploaded_file is not None:
try:
# Attempt to read the uploaded file as a DataFrame
df = pd.read_csv(uploaded_file).head(5)

# Display the DataFrame in the app
st.write("Data successfully uploaded and read as DataFrame:")
st.dataframe(df)

# Store the DataFrame in the session state
st.session_state.uploaded_data = df
except Exception as e:
st.error(f"Error reading the file: {e}")
```

```
def display_user_request_input():
if "show_request_input" not in st.session_state:
st.session_state.show_request_input = True
if st.session_state.show_request_input:
if st.session_state.get("previous_user_request") != st.session_state.get("user_request", ""):
st.session_state.previous_user_request = st.session_state.get("user_request", "")
if st.session_state.get("user_request", ""):
handle_user_request(st.session_state)
else:
st.session_state.agents = []
st.session_state.show_request_input = False
st.experimental_rerun()


def extract_code_from_response(response):
code_pattern = r"```(.*?)```"
code_blocks = re.findall(code_pattern, response, re.DOTALL)

html_pattern = r"<html.*?>.*?</html>"
html_blocks = re.findall(html_pattern, response, re.DOTALL | re.IGNORECASE)

js_pattern = r"<script.*?>.*?</script>"
js_blocks = re.findall(js_pattern, response, re.DOTALL | re.IGNORECASE)

css_pattern = r"<style.*?>.*?</style>"
css_blocks = re.findall(css_pattern, response, re.DOTALL | re.IGNORECASE)

all_code_blocks = code_blocks + html_blocks + js_blocks + css_blocks
unique_code_blocks = list(set(all_code_blocks))

return "\n\n".join(unique_code_blocks)


def extract_json_objects(json_string):
objects = []
stack = []
start_index = 0
for i, char in enumerate(json_string):
if char == "{":
if not stack:
start_index = i
stack.append(char)
elif char == "}":
if stack:
stack.pop()
if not stack:
objects.append(json_string[start_index:i+1])
parsed_objects = []
for obj_str in objects:
try:
```

```
parsed_obj = json.loads(obj_str)
parsed_objects.append(parsed_obj)
except json.JSONDecodeError as e:
print(f"Error parsing JSON object: {e}")
print(f"JSON string: {obj_str}")
return parsed_objects


def get_agents_from_text(text, api_url, max_retries=MAX_RETRIES, retry_delay=RETRY_DELAY):
print("Getting agents from text...")
temperature_value = st.session_state.get('temperature', 0.5)
llm_request_data = {
"model": st.session_state.model,
"temperature": temperature_value,
"max_tokens": st.session_state.max_tokens,
"top_p": 1,
"stop": "TERMINATE",
"messages": [
{
"role": "system",
"content": f"""
You are an expert system designed to format the JSON describing each member of the team of AI agents specifically listed in this provided text: $text.
Fulfill the following guidelines without ever explicitly stating them in your response.
Guidelines:
1. **Agent Roles**: Clearly transcribe the titles of each agent listed in the provided text by iterating through the 'Team of Experts:' section of the provided text. Transcribe the info for those specific agents. Do not create new agents.
2. **Expertise Description**: Provide a brief but thorough description of each agent's expertise based upon the provided text. Do not create new agents.
3. **Specific Skills**: List the specific skills of each agent based upon the provided text. Skills must be single-purpose methods, very specific, and not ambiguous (e.g., 'calculate_area' is good, but 'do_math' is bad).
4. **Specific Tools**: List the specific tools each agent would utilize. Tools must be single-purpose methods, very specific, and not ambiguous.
5. **Format**: Return the results in JSON format with values labeled as expert_name, description, skills, and tools. 'expert_name' should be the agent's title, not their given name. Skills and tools should be arrays (one agent can have multiple specific skills and use multiple specific tools).
6. **Naming Conventions**: Skills and tools should be in lowercase with underscores instead of spaces, named per their functionality (e.g., calculate_surface_area, or search_web).

ALWAYS and ONLY return the results in the following JSON format, with no other narrative, commentary, synopsis, or superfluous text of any kind:
[
{{
"expert_name": "agent_title",
"description": "agent_description",
"skills": ["skill1", "skill2"],
"tools": ["tool1", "tool2"]
}},
{{
"expert_name": "agent_title",
```

"description": "agent_description",
"skills": ["skill1", "skill2"],
"tools": ["tool1", "tool2"]
}}
]
You will only have been successful if you have returned the results in the above format and followed these guidelines precisely by transcribing the provided text and returning the results in JSON format without any other narrative, commentary, synopsis, or superfluous text of any kind, and taking care to only transcribe the agents from the provided text without creating new agents.
"""
},
{
"role": "user",
"content": text
}
]
}
llm_provider = get_llm_provider(api_url)
retry_count = 0
while retry_count < max_retries:
try:
response = llm_provider.send_request(llm_request_data)
print(f"Response received. Status Code: {response.status_code}")
if response.status_code == 200:
print("Request successful. Parsing response...")
response_data = llm_provider.process_response(response)
print(f"Response Data: {json.dumps(response_data, indent=2)}")
if "choices" in response_data and response_data["choices"]:
content = response_data["choices"][0]["message"]["content"]
print(f"Content: {content}")
try:
json_data = json.loads(content)
if isinstance(json_data, list):
autogen_agents = []
crewai_agents = []
for agent_data in json_data:
expert_name = agent_data.get('expert_name', '')
if not expert_name:
print("Missing agent name. Retrying...")
retry_count += 1
time.sleep(retry_delay)
continue
description = agent_data.get('description', '')
skills = agent_data.get('skills', [])
tools = agent_data.get('tools', [])
agent_skills = st.session_state.selected_skills
autogen_agent_data = {
"type": "assistant",
"config": {
"name": expert_name,

```
"llm_config": {
"config_list": [
{
"user_id": "default",
"timestamp": datetime.datetime.now().isoformat(),
"model": st.session_state.model,
"base_url": None,
"api_type": None,
"api_version": None,
"description": "OpenAI model configuration"
}
],
"temperature": temperature_value,
"timeout": 600,
"cache_seed": 42
},
"human_input_mode": "NEVER",
"max_consecutive_auto_reply": 8,
"system_message": f"You are a helpful assistant that can act as {expert_name} who {description}."
},
"description": description,
"skills": agent_skills,
"tools": tools
}
crewai_agent_data = {
"name": expert_name,
"description": description,
"skills": agent_skills,
"tools": tools,
"verbose": True,
"allow_delegation": True
}
autogen_agents.append(autogen_agent_data)
crewai_agents.append(crewai_agent_data)
print(f"AutoGen Agents: {autogen_agents}")
print(f"CrewAI Agents: {crewai_agents}")
return autogen_agents, crewai_agents
else:
print("Invalid JSON format. Expected a list of agents.")
return [], []
except json.JSONDecodeError as e:
print(f"Error parsing JSON: {e}")
print(f"Content: {content}")
json_data = extract_json_objects(content)
if json_data:
autogen_agents = []
crewai_agents = []
for agent_data in json_data:
expert_name = agent_data.get('expert_name', '')
if not expert_name:
```

```
print("Missing agent name. Retrying...")
retry_count += 1
time.sleep(retry_delay)
continue
description = agent_data.get('description', '')
skills = agent_data.get('skills', [])
tools = agent_data.get('tools', [])
agent_skills = st.session_state.selected_skills
autogen_agent_data = {
"type": "assistant",
"config": {
"name": expert_name,
"llm_config": {
"config_list": [
{
"user_id": "default",
"timestamp": datetime.datetime.now().isoformat(),
"model": st.session_state.model,
"base_url": None,
"api_type": None,
"api_version": None,
"description": "OpenAI model configuration"
}
],
"temperature": temperature_value,
"timeout": 600,
"cache_seed": 42
},
"human_input_mode": "NEVER",
"max_consecutive_auto_reply": 8,
"system_message": f"You are a helpful assistant that can act as {expert_name} who {description}."
},
"description": description,
"skills": agent_skills,
"tools": tools
}
crewai_agent_data = {
"name": expert_name,
"description": description,
"skills": agent_skills,
"tools": tools,
"verbose": True,
"allow_delegation": True
}
autogen_agents.append(autogen_agent_data)
crewai_agents.append(crewai_agent_data)
print(f"AutoGen Agents: {autogen_agents}")
print(f"CrewAI Agents: {crewai_agents}")
return autogen_agents, crewai_agents
else:
```

```
print("Failed to extract JSON objects from content.")
return [], []
else:
print("No agents data found in response")
else:
print(f"API request failed with status code {response.status_code}: {response.text}")
except Exception as e:
print(f"Error making API request: {e}")
retry_count += 1
time.sleep(retry_delay)
print(f"Maximum retries ({max_retries}) exceeded. Failed to retrieve valid agent names.")
return [], []


def get_discussion_history():
if "discussion_history" not in st.session_state:
st.session_state.discussion_history = ""
return st.session_state.discussion_history


def get_workflow_from_agents(agents):
current_timestamp = datetime.datetime.now().isoformat()
temperature_value = st.session_state.get('temperature', 0.3)

workflow = {
"name": "AutoGroq Workflow",
"description": "Workflow auto-generated by AutoGroq.",
"sender": {
"type": "userproxy",
"config": {
"name": "userproxy",
"llm_config": False,
"human_input_mode": "NEVER",
"max_consecutive_auto_reply": 5,
"system_message": "You are a helpful assistant.",
"is_termination_msg": None,
"code_execution_config": {
"work_dir": None,
"use_docker": False
},
"default_auto_reply": "",
"description": None
},
"timestamp": current_timestamp,
"user_id": "default",
"skills": []
},
"receiver": {
"type": "groupchat",
"config": {
```

"name": "group_chat_manager",
"llm_config": {
"config_list": [
{
"user_id": "default",
"timestamp": datetime.datetime.now().isoformat(),
"model": st.session_state.model,
"base_url": None,
"api_type": None,
"api_version": None,
"description": "OpenAI model configuration"
}
],
"temperature": temperature_value,
"cache_seed": 42,
"timeout": 600,
"max_tokens": MODEL_TOKEN_LIMITS.get(st.session_state.model, 4096),
"extra_body": None
},
"human_input_mode": "NEVER",
"max_consecutive_auto_reply": 10,
"system_message": "Group chat manager",
"is_termination_msg": None,
"code_execution_config": None,
"default_auto_reply": "",
"description": None
},
"groupchat_config": {
"agents": [],
"admin_name": "Admin",
"messages": [],
"max_round": 10,
"speaker_selection_method": "auto",
"allow_repeat_speaker": True
},
"timestamp": current_timestamp,
"user_id": "default",
"skills": []
},
"type": "groupchat",
"user_id": "default",
"timestamp": current_timestamp,
"summary_method": "last"
}

for index, agent in enumerate(agents):
agent_name = agent["config"]["name"]
description = agent["description"]
formatted_agent_name = sanitize_text(agent_name).lower().replace(' ', '_')
sanitized_description = sanitize_text(description)

```
system_message = f"You are a helpful assistant that can act as {agent_name} who {sanitized_description}."
if index == 0:
other_agent_names = [sanitize_text(a['config']['name']).lower().replace(' ', '_') for a in agents[1:] if a in st.session_state.agents]  # Filter out deleted agents
system_message += f" You are the primary coordinator who will receive suggestions or advice from all the other agents ({', '.join(other_agent_names)}). You must ensure that the final response integrates the suggestions from other agents or team members. YOUR FINAL RESPONSE MUST OFFER THE COMPLETE RESOLUTION TO THE USER'S REQUEST. When the user's request has been satisfied and all perspectives are integrated, you can respond with TERMINATE."

other_agent_names = [sanitize_text(a['config']['name']).lower().replace(' ', '_') for a in agents[1:]]
system_message += f" You are the primary coordinator who will receive suggestions or advice from all the other agents ({', '.join(other_agent_names)}). You must ensure that the final response integrates the suggestions from other agents or team members. YOUR FINAL RESPONSE MUST OFFER THE COMPLETE RESOLUTION TO THE USER'S REQUEST. When the user's request has been satisfied and all perspectives are integrated, you can respond with TERMINATE."

agent_config = {
"type": "assistant",
"config": {
"name": formatted_agent_name,
"llm_config": {
"config_list": [
{
"user_id": "default",
"timestamp": datetime.datetime.now().isoformat(),
"model": st.session_state.model,
"base_url": None,
"api_type": None,
"api_version": None,
"description": "OpenAI model configuration"
}
],
"temperature": temperature_value,
"cache_seed": 42,
"timeout": 600,
"max_tokens": MODEL_TOKEN_LIMITS.get(st.session_state.model, 4096),
"extra_body": None
},
"human_input_mode": "NEVER",
"max_consecutive_auto_reply": 8,
"system_message": system_message,
"is_termination_msg": None,
"code_execution_config": None,
"default_auto_reply": "",
"description": None
},
"timestamp": current_timestamp,
"user_id": "default",
```

```
"skills": []  # Set skills to null only in the workflow JSON
}

workflow["receiver"]["groupchat_config"]["agents"].append(agent_config)

crewai_agents = []
for agent in agents:
if agent not in st.session_state.agents:  # Check if the agent exists in st.session_state.agents
continue  # Skip the agent if it has been deleted

_, crewai_agent_data = create_agent_data(agent)
crewai_agents.append(crewai_agent_data)

return workflow, crewai_agents


def handle_user_request(session_state):
print("Debug: Handling user request for session state: ", session_state)
user_request = session_state.user_request
max_retries = MAX_RETRIES
retry_delay = RETRY_DELAY

for retry in range(max_retries):
try:
print("Debug: Sending request to rephrase_prompt")
rephrased_text = rephrase_prompt(user_request, API_URL)  # Pass the API_URL to rephrase_prompt
print(f"Debug: Rephrased text: {rephrased_text}")
if rephrased_text:
session_state.rephrased_request = rephrased_text
break  # Exit the loop if successful
else:
print("Error: Failed to rephrase the user request.")
st.warning("Failed to rephrase the user request. Please try again.")
return  # Exit the function if rephrasing fails
except Exception as e:
print(f"Error occurred in handle_user_request: {str(e)}")
if retry < max_retries - 1:
print(f"Retrying in {retry_delay} second(s)...")
time.sleep(retry_delay)
else:
print("Max retries exceeded.")
st.warning("An error occurred. Please try again.")
return  # Exit the function if max retries are exceeded

if "rephrased_request" not in session_state:
st.warning("Failed to rephrase the user request. Please try again.")
return

rephrased_text = session_state.rephrased_request
```

```
if "project_manager_output" not in session_state:
# Create the Project Manager agent only if it hasn't been created before
project_manager_output = create_project_manager(rephrased_text, API_URL)

if not project_manager_output:
print("Error: Failed to create Project Manager.")
st.warning("Failed to create Project Manager. Please try again.")
return

session_state.project_manager_output = project_manager_output

# Create an instance of the Current_Project class
current_project = Current_Project()
current_project.set_re_engineered_prompt(rephrased_text)

# Extract objectives and deliverables from the project manager's output
objectives_pattern = r"Objectives:\n(.*?)(?=Deliverables|$)"
deliverables_pattern = r"Deliverables:\n(.*?)(?=Timeline|Team of Experts|$)"

objectives_match = re.search(objectives_pattern, project_manager_output, re.DOTALL)
if objectives_match:
objectives = objectives_match.group(1).strip().split("\n")
for objective in objectives:
current_project.add_objective(objective.strip())

deliverables_match = re.search(deliverables_pattern, project_manager_output, re.DOTALL)
if deliverables_match:
deliverables = deliverables_match.group(1).strip().split("\n")
for deliverable in deliverables:
current_project.add_deliverable(deliverable.strip())

session_state.current_project = current_project

# Update the discussion and whiteboard with the Project Manager's initial response
update_discussion_and_whiteboard("Project Manager", project_manager_output, "")
else:
# Retrieve the previously created Project Manager's output from the session state
project_manager_output = session_state.project_manager_output

team_of_experts_pattern = r"Team of Experts:\n(.*)"
match = re.search(team_of_experts_pattern, project_manager_output, re.DOTALL)
if match:
team_of_experts_text = match.group(1).strip()
else:
print("Error: 'Team of Experts' section not found in Project Manager's output.")
st.warning("Failed to extract the team of experts from the Project Manager's output. Please try again.")
return

autogen_agents, crewai_agents = get_agents_from_text(team_of_experts_text, API_URL)
```

```
print(f"Debug: AutoGen Agents: {autogen_agents}")
print(f"Debug: CrewAI Agents: {crewai_agents}")

if not autogen_agents:
print("Error: No agents created.")
st.warning("Failed to create agents. Please try again.")
return

# Set the agents attribute in the session state
session_state.agents = autogen_agents

workflow_data, _ = get_workflow_from_agents(autogen_agents)
print(f"Debug: Workflow data: {workflow_data}")
print(f"Debug: CrewAI agents: {crewai_agents}")

autogen_zip_buffer, crewai_zip_buffer = zip_files_in_memory(workflow_data)
session_state.autogen_zip_buffer = autogen_zip_buffer
session_state.crewai_zip_buffer = crewai_zip_buffer


def load_skill_functions():
script_dir = os.path.dirname(os.path.abspath(__file__))
skill_folder = os.path.join(script_dir, "skills")
skill_files = [f for f in os.listdir(skill_folder) if f.endswith(".py")]
skill_functions = {}
for skill_file in skill_files:
skill_name = os.path.splitext(skill_file)[0]
skill_module = importlib.import_module(f"skills.{skill_name}")
if hasattr(skill_module, skill_name):
skill_functions[skill_name] = getattr(skill_module, skill_name)
st.session_state.skill_functions = skill_functions


def regenerate_json_files_and_zip():
# Get the updated workflow data
workflow_data, _ = get_workflow_from_agents(st.session_state.agents)

# Regenerate the zip files
autogen_zip_buffer, crewai_zip_buffer = zip_files_in_memory(workflow_data)

# Update the zip buffers in the session state
st.session_state.autogen_zip_buffer = autogen_zip_buffer
st.session_state.crewai_zip_buffer = crewai_zip_buffer


def regenerate_zip_files():
if "agents" in st.session_state:
workflow_data, _ = get_workflow_from_agents(st.session_state.agents)
autogen_zip_buffer, crewai_zip_buffer = zip_files_in_memory(workflow_data)
st.session_state.autogen_zip_buffer = autogen_zip_buffer
```

```
st.session_state.crewai_zip_buffer = crewai_zip_buffer
print("Zip files regenerated.")
else:
print("No agents found. Skipping zip file regeneration.")


def rephrase_prompt(user_request, api_url):
temperature_value = st.session_state.get('temperature', 0.1)
print("Executing rephrase_prompt()")
print(f"Debug: api_url: {api_url}")

refactoring_prompt = f"""
Act as a professional prompt engineer and efactor the following user request into an optimized prompt. Your goal is to rephrase the request with a focus on the satisfying all following the criteria without explicitly stating them:
1. Clarity: Ensure the prompt is clear and unambiguous.
2. Specific Instructions: Provide detailed steps or guidelines.
3. Context: Include necessary background information.
4. Structure: Organize the prompt logically.
5. Language: Use concise and precise language.
6. Examples: Offer examples to illustrate the desired output.
7. Constraints: Define any limits or guidelines.
8. Engagement: Make the prompt engaging and interesting.
9. Feedback Mechanism: Suggest a way to improve or iterate on the response.
Do NOT reply with a direct response to these instructions OR the original user request. Instead, rephrase the user's request as a well-structured prompt, and
return ONLY that rephrased prompt. Do not preface the rephrased prompt with any other text or superfluous narrative. Do not enclose the rephrased prompt in quotes. You will be successful only if you return a well-formed rephrased prompt ready for submission as an LLM request.
User request: "{user_request}"
Rephrased:
"""

model = st.session_state.model
max_tokens = MODEL_TOKEN_LIMITS.get(model, 4096)  # Use the appropriate max_tokens value based on the selected model

llm_request_data = {
"model": model,
"temperature": temperature_value,
"max_tokens": max_tokens,
"top_p": 1,
"stop": "TERMINATE",
"messages": [
{
"role": "user",
"content": refactoring_prompt,
},
],
}
```

```
llm_provider = get_llm_provider(api_url)  # Pass the api_url to get_llm_provider

try:
print("Sending request to LLM API...")
print(f"Request Details:")
print(f" URL: {api_url}")  # Print the API URL
print(f" Model: {model}")
print(f" Max Tokens: {max_tokens}")
print(f" Temperature: {temperature_value}")
print(f" Messages: {llm_request_data['messages']}")

response = llm_provider.send_request(llm_request_data)
print(f"Response received. Status Code: {response.status_code}")
print(f"Response Content: {response.text}")

if response.status_code == 200:
print("Request successful. Parsing response...")
response_data = llm_provider.process_response(response)
print(f"Response Data: {json.dumps(response_data, indent=2)}")

if "choices" in response_data and len(response_data["choices"]) > 0:
rephrased = response_data["choices"][0]["message"]["content"]
return rephrased.strip()
else:
print("Error: Unexpected response format. 'choices' field missing or empty.")
return None
else:
print(f"Request failed. Status Code: {response.status_code}")
print(f"Response Content: {response.text}")
return None
except Exception as e:
print(f"An error occurred: {str(e)}")
return None



def update_discussion_and_whiteboard(agent_name, response, user_input):
if user_input:
user_input_text = f"\n\n\n\n{user_input}\n\n"
st.session_state.discussion_history += user_input_text

if "last_agent" not in st.session_state or st.session_state.last_agent != agent_name:
response_text = f"{agent_name}:\n\n{response}\n\n===\n\n"
else:
response_text = f"{response}\n\n===\n\n"

st.session_state.discussion_history += response_text
code_blocks = extract_code_from_response(response)
st.session_state.whiteboard = code_blocks
st.session_state.last_agent = agent_name
st.session_state.last_comment = response_text
```

```
def zip_files_in_memory(workflow_data):
autogen_zip_buffer = io.BytesIO()
crewai_zip_buffer = io.BytesIO()
skill_folder = os.path.join(os.path.dirname(os.path.abspath(__file__)), "skills")
autogen_file_data = {}
for agent in st.session_state.agents:
agent_name = agent['config']['name']
formatted_agent_name = sanitize_text(agent_name).lower().replace(' ', '_')
agent_file_name = f"{formatted_agent_name}.json"

# Use the agent-specific model configuration
autogen_agent_data, _ = create_agent_data(agent)
autogen_agent_data['config']['name'] = formatted_agent_name
autogen_agent_data['config']['llm_config']['config_list'][0]['model'] = agent['config']['llm_config']['config_list'][0]['model']
autogen_agent_data['config']['llm_config']['max_tokens'] = agent['config']['llm_config'].get('max_tokens',
MODEL_TOKEN_LIMITS.get(st.session_state.model, 4096))
autogen_agent_data['skills'] = []

for skill_name in st.session_state.selected_skills:
skill_file_path = os.path.join(skill_folder, f"{skill_name}.py")
with open(skill_file_path, 'r') as file:
skill_data = file.read()
skill_json = create_skill_data(skill_data)
autogen_agent_data['skills'].append(skill_json)
agent_file_data = "# Created by AutoGroq™ [https://github.com/jgravelle/AutoGroq]\n# https://j.gravelle.us\n\n"
agent_file_data += json.dumps(autogen_agent_data, indent=2)
agent_file_data = agent_file_data.encode('utf-8')
autogen_file_data[f"agents/{agent_file_name}"] = agent_file_data
for skill_name in st.session_state.selected_skills:
skill_file_path = os.path.join(skill_folder, f"{skill_name}.py")
with open(skill_file_path, 'r') as file:
skill_data = file.read()
skill_json = "# Created by AutoGroq™ [https://github.com/jgravelle/AutoGroq]\n# https://j.gravelle.us\n\n"
skill_json += json.dumps(create_skill_data(skill_data), indent=2)
skill_json = skill_json.encode('utf-8')
autogen_file_data[f"skills/{skill_name}.json"] = skill_json
workflow_file_name = "workflow.json"
workflow_file_data = "# Created by AutoGroq™ [https://github.com/jgravelle/AutoGroq]\n# https://j.gravelle.us\n\n"
workflow_file_data += json.dumps(workflow_data, indent=2)
workflow_file_data = workflow_file_data.encode('utf-8')
autogen_file_data[workflow_file_name] = workflow_file_data
crewai_file_data = {}
for index, agent in enumerate(st.session_state.agents):
agent_name = agent['config']['name']
formatted_agent_name = sanitize_text(agent_name).lower().replace(' ', '_')
crewai_agent_data = create_agent_data(agent)[1]
crewai_agent_data['name'] = formatted_agent_name
agent_file_name = f"{formatted_agent_name}.json"
```

```
agent_file_data = "# Created by AutoGroq™ [https://github.com/jgravelle/AutoGroq]\n# https://j.gravelle.us\n\n"
agent_file_data += json.dumps(crewai_agent_data, indent=2)
agent_file_data = agent_file_data.encode('utf-8')
crewai_file_data[f"agents/{agent_file_name}"] = agent_file_data
create_zip_file(autogen_zip_buffer, autogen_file_data)
create_zip_file(crewai_zip_buffer, crewai_file_data)
autogen_zip_buffer.seek(0)
crewai_zip_buffer.seek(0)
return autogen_zip_buffer, crewai_zip_buffer
```

## base_provider.py

```
# llm_providers/base_provider.py
from abc import ABC, abstractmethod

class BaseLLMProvider(ABC):
@abstractmethod
def send_request(self, data):
pass

@abstractmethod
def process_response(self, response):
pass
```

## groq_provider.py

```
import json
import requests
from auth_utils import get_api_key
from llm_providers.base_provider import BaseLLMProvider

class GroqProvider(BaseLLMProvider):
def __init__(self, api_url):
self.api_key = get_api_key()
self.api_url = api_url


def send_request(self, data):
headers = {
"Authorization": f"Bearer {self.api_key}",
"Content-Type": "application/json",
}
# Ensure data is a JSON string
if isinstance(data, dict):
json_data = json.dumps(data)
else:
json_data = data
response = requests.post(self.api_url, data=json_data, headers=headers)
return response
```

```
def process_response(self, response):
if response.status_code == 200:
return response.json()
else:
raise Exception(f"Request failed with status code {response.status_code}")
```

## lmstudio_provider.py

```
import json
import requests
from llm_providers.base_provider import BaseLLMProvider

class LmstudioProvider(BaseLLMProvider):
def __init__(self, api_url):
self.api_url = api_url


def send_request(self, data):
headers = {
"Content-Type": "application/json",
}

# Construct the request data in the format expected by the LM Studio API
lm_studio_request_data = {
"model": data["model"],
"messages": data["messages"],
"temperature": data.get("temperature", 0.1),
"max_tokens": data.get("max_tokens", 2048),
"stop": data.get("stop", "TERMINATE"),
}

# Ensure data is a JSON string
if isinstance(lm_studio_request_data, dict):
json_data = json.dumps(lm_studio_request_data)
else:
json_data = lm_studio_request_data

response = requests.post(f"{self.api_url}", data=json_data, headers=headers)
return response

def process_response(self, response):
if response.status_code == 200:
response_data = response.json()
if "choices" in response_data:
content = response_data["choices"][0]["message"]["content"]
return {
"choices": [
{
"message": {
"content": content.strip()
```

```
}
}
]
}
else:
raise Exception("Unexpected response format. 'choices' field missing.")
else:
raise Exception(f"Request failed with status code {response.status_code}")
```

## ollama_provider.py

```
import json
import requests
from llm_providers.base_provider import BaseLLMProvider

class OllamaProvider(BaseLLMProvider):
def __init__(self, api_url):
self.api_url = api_url


def send_request(self, data):
headers = {
"Content-Type": "application/json",
}
# Construct the request data in the format expected by the Ollama API
ollama_request_data = {
"model": data["model"],
"prompt": data["messages"][0]["content"],
"temperature": data.get("temperature", 0.1),
"max_tokens": data.get("max_tokens", 2048),
"stop": data.get("stop", "TERMINATE"),
"stream": False,
}
# Ensure data is a JSON string
if isinstance(ollama_request_data, dict):
json_data = json.dumps(ollama_request_data)
else:
json_data = ollama_request_data
response = requests.post(self.api_url, data=json_data, headers=headers)
return response


def process_response(self, response):
if response.status_code == 200:
response_data = response.json()
if "response" in response_data:
content = response_data["response"].strip()
if content:
return {
"choices": [
```

```
{
"message": {
"content": content
}
}
]
}
else:
raise Exception("Empty response received from the Ollama API.")
else:
raise Exception("Unexpected response format. 'response' field missing.")
else:
raise Exception(f"Request failed with status code {response.status_code}")
```

## openai_provider.py

```
# Thanks to NeutrinoTek:  https://github.com/neutrinotek

import requests
import json
from auth_utils import get_api_key
from llm_providers.base_provider import BaseLLMProvider

class OpenaiProvider(BaseLLMProvider):

def __init__(self, api_url):
self.api_key = get_api_key()
self.api_url = api_url


def send_request(self, data):
headers = {
"Authorization": f"Bearer {self.api_key}",
"Content-Type": "application/json",
}

# Ensure data is a JSON string
if isinstance(data, dict):
json_data = json.dumps(data)
else:
json_data = data

response = requests.post(self.api_url, data=json_data, headers=headers)
return response


def process_response(self, response):
if response.status_code == 200:
return response.json()
else:
raise Exception(f"Request failed with status code {response.status_code}")
```

## document_indexer.py

```
#  Thanks to MADTANK:  https://github.com/madtank
#  README:  https://github.com/madtank/autogenstudio-skills/blob/main/rag/README.md

import argparse
import csv
import json
import os
import pickle
import re
import traceback
from typing import Dict, List, Literal, Tuple

try:
import tiktoken
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
except ImportError:
raise ImportError("Please install the dependencies first.")


def chunk_str_overlap(
s: str,
separator: chr = "\n",
num_tokens: int = 64,
step_tokens: int = 64,
encoding: tiktoken.Encoding = None,
) -> List[str]:
"""
Split a string into chunks with overlap
:param s: the input string
:param separator: the separator to split the string
:param num_tokens: the number of tokens in each chunk
:param step_tokens: the number of tokens to step forward
:param encoding: the encoding to encode the string
"""
assert step_tokens <= num_tokens, (
f"The number of tokens {num_tokens} in each chunk " f"should be larger than the step size {step_tokens}."
)

lines = s.split(separator)
chunks = dict()
final_chunks = []

if len(lines) == 0:
return []

first_line = lines[0]
first_line_size = len(encoding.encode(first_line))
```

```
chunks[0] = [first_line, first_line_size]

this_step_size = first_line_size

for i in range(1, len(lines)):
line = lines[i]
line_size = len(encoding.encode(line))

to_pop = []
for key in chunks:
if chunks[key][1] + line_size > num_tokens:
to_pop.append(key)
else:
chunks[key][0] += f"{separator}{line}"
chunks[key][1] += line_size
final_chunks += [chunks.pop(key)[0] for key in to_pop]

if this_step_size + line_size > step_tokens:
chunks[i] = [line, line_size]
this_step_size = 0
this_step_size += line_size

max_remained_chunk = ""
max_remained_chunk_size = 0
for key in chunks:
if chunks[key][1] > max_remained_chunk_size:
max_remained_chunk_size = chunks[key][1]
max_remained_chunk = chunks[key][0]
if max_remained_chunk_size > 0:
final_chunks.append(max_remained_chunk)

return final_chunks


def get_title(
file_name: str,
prop="title: ",
) -> str:
"""
Get the title of a file
:param file_name: the file name
:param prop: the property to get the title
"""
with open(file_name, encoding="utf-8", errors="ignore") as f_in:
for line in f_in:
line = line.strip()
if line and (line.startswith(prop) or any([c.isalnum() for c in line])):
return line
return ""
```

```
def extract_text_from_file(
file: str,
file_type: Literal["pdf", "docx", "csv", "pptx"],
) -> Tuple[str, str]:
"""
Extract text from a file in pdf, docx, csv or pptx format
:param file: the file path
:param file_type: the extension of the file
"""
if file_type == "pdf":
try:
from pypdf import PdfReader
except ImportError:
raise ImportError("Please install pypdf first.")
# Extract text from pdf using PyPDF2
reader = PdfReader(file)
extracted_text = " ".join([page.extract_text() for page in reader.pages])
title = extracted_text.split("\n")[0]
elif file_type == "docx":
try:
import docx2txt
except ImportError:
raise ImportError("Please install docx2txt first.")
# Extract text from docx using docx2txt
extracted_text = docx2txt.process(file)
title = extracted_text.split("\n")[0]
elif file_type == "csv":
# Extract text from csv using csv module
extracted_text = ""
title = ""
reader = csv.reader(file)
for row in reader:
extracted_text += " ".join(row) + "\n"
elif file_type == "pptx":
try:
import pptx
except ImportError:
raise ImportError("Please install python-pptx first.")
extracted_text = ""
no_title = True
title = ""
presentation = pptx.Presentation(file)
for slide in presentation.slides:
for shape in slide.shapes:
if shape.has_text_frame:
for paragraph in shape.text_frame.paragraphs:
for run in paragraph.runs:
extracted_text += run.text + " "
if no_title and len(run.text) > 10:
```

```
title = run.text
no_title = False
extracted_text += "\n"
else:
# Unsupported file type
raise ValueError(f"Unsupported file type: {file_type}")

return title[:100], extracted_text



def text_parser(
read_file: str,
) -> Tuple[str, str]:
"""
Returns the title, parsed text and a BeautifulSoup object with different file extension
: param read_file: the input file with a given extension
: return: the title, parsed text and a BeautifulSoup object, the BeautifulSoup object is used to get the document
link from the html files
"""
filename, extension = os.path.splitext(read_file)
extension = extension.lstrip(".")
title = filename
soup = None
supported_extensions = ["md", "markdown", "html", "htm", "txt", "json", "jsonl"]
other_extensions = ["docx", "pptx", "pdf", "csv"]

# utf-8-sig will treat BOM header as a metadata of a file not a part of the file content
default_encoding = "utf-8-sig"

if extension in ("md", "markdown", "txt"):
title = get_title(read_file)
with open(read_file, "r", encoding=default_encoding, errors="ignore") as f:
text = f.read()
elif extension in ("html", "htm"):
from bs4 import BeautifulSoup

with open(read_file, "r", encoding=default_encoding, errors="ignore") as f:
soup = BeautifulSoup(f, "html.parser")
title = next(soup.stripped_strings)[:100]
text = soup.get_text("\n")
# read json/jsonl file in and convert each json to a row of string
elif extension in ("json", "jsonl"):
try:
with open(read_file, "r", encoding=default_encoding, errors="ignore") as f:
data = json.load(f) if extension == "json" else [json.loads(line) for line in f]
except:
# json file encoding issue, skip this file
return title, ""

if isinstance(data, dict):
```

```
text = json.dumps(data)
elif isinstance(data, list):
content_list = [json.dumps(each_json) for each_json in data]
text = "\n".join(content_list)
title = filename
elif extension in other_extensions:
title, text = extract_text_from_file(read_file, extension)
else:  # no support for other format
print(
f"Not support for file with extension: {extension}. "
f"The supported extensions are {supported_extensions}",
)
return title, ""

output_text = re.sub(r"\n{3,}", "\n\n", text)
# keep whitespaces for formatting
output_text = re.sub(r"-{3,}", "---", output_text)
output_text = re.sub(r"\*{3,}", "***", output_text)
output_text = re.sub(r"_{3,}", "___", output_text)

return title, output_text



def chunk_document(
doc_path: str,
chunk_size: int,
chunk_step: int,
) -> Tuple[int, List[str], List[Dict[str, str]], Dict[str, int]]:
"""
Split documents into chunks
:param doc_path: the path of the documents
:param chunk_size: the size of the chunk
:param chunk_step: the step size of the chunk
"""
texts = []
metadata_list = []
file_count = 0
chunk_id_to_index = dict()

enc = tiktoken.encoding_for_model("gpt-3.5-turbo")

# traverse all files under dir
print("Split documents into chunks...")
for root, dirs, files in os.walk(doc_path):
for name in files:
f = os.path.join(root, name)
print(f"Reading {f}")
try:
title, content = text_parser(f)
file_count += 1
```

```
if file_count % 100 == 0:
print(f"{file_count} files read.")

if len(content) == 0:
continue

chunks = chunk_str_overlap(
content.strip(),
num_tokens=chunk_size,
step_tokens=chunk_step,
separator="\n",
encoding=enc,
)
source = os.path.sep.join(f.split(os.path.sep)[4:])
for i in range(len(chunks)):
# custom metadata if needed
metadata = {
"source": source,
"title": title,
"chunk_id": i,
}
chunk_id_to_index[f"{source}_{i}"] = len(texts) + i
metadata_list.append(metadata)
texts.extend(chunks)
except Exception as e:
print(f"Error encountered when reading {f}: {traceback.format_exc()} {e}")
return file_count, texts, metadata_list, chunk_id_to_index


if __name__ == "__main__":
# parse arguments
parser = argparse.ArgumentParser()
parser.add_argument(
"-d",
"--doc_path",
help="the path of the documents",
type=str,
default="documents",
)
parser.add_argument(
"-c",
"--chunk_size",
help="the size of the chunk",
type=int,
default=64,
)
parser.add_argument(
"-s",
"--chunk_step",
help="the step size of the chunk",
```

```
type=int,
default=64,
)
parser.add_argument(
"-o",
"--output_path",
help="the path of the output",
type=str,
default="knowledge",
)
args = parser.parse_args()

file_count, texts, metadata_list, chunk_id_to_index = chunk_document(
doc_path=args.doc_path,
chunk_size=args.chunk_size,
chunk_step=args.chunk_step,
)
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
vectorstore = FAISS.from_texts(
texts=texts,
metadatas=metadata_list,
embedding=embeddings,
)
vectorstore.save_local(folder_path=args.output_path)
with open(os.path.join(args.output_path, "chunk_id_to_index.pkl"), "wb") as f:
pickle.dump(chunk_id_to_index, f)
print(f"Saved vectorstore to {args.output_path}")
```

## document_retriever.py

```
#  Thanks to MADTANK:  https://github.com/madtank
#  README:  https://github.com/madtank/autogenstudio-skills/blob/main/rag/README.md

import os
import pickle
import json
import argparse

try:
import tiktoken
from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores import FAISS
except ImportError:
raise ImportError("Please install langchain-community first.")

# Configuration - Users/AI skill developers must update this path to their specific index folder
# To test with sample data set index_folder to "knowledge"
CONFIG = {
"index_folder": "rag/knowledge",  # TODO: Update this path before using
}
```

```
class DocumentRetriever:
def __init__(self, index_folder):
self.index_folder = index_folder
self.vectorstore = None
self.chunk_id_to_index = None
self.embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
self._init()
self.enc = tiktoken.encoding_for_model("gpt-3.5-turbo")

def _init(self):
self.vectorstore = FAISS.load_local(
folder_path=self.index_folder,
embeddings=self.embeddings,
)
with open(os.path.join(self.index_folder, "chunk_id_to_index.pkl"), "rb") as f:
self.chunk_id_to_index = pickle.load(f)

def __call__(self, query: str, size: int = 5, target_length: int = 256):
if self.vectorstore is None:
raise Exception("Vectorstore not initialized")

result = self.vectorstore.similarity_search(query=query, k=size)
expanded_chunks = self.do_expand(result, target_length)

return json.dumps(expanded_chunks, indent=4)

def do_expand(self, result, target_length):
expanded_chunks = []
# do expansion
for r in result:
source = r.metadata["source"]
chunk_id = r.metadata["chunk_id"]
content = r.page_content

expanded_result = content
left_chunk_id, right_chunk_id = chunk_id - 1, chunk_id + 1
left_valid, right_valid = True, True
chunk_ids = [chunk_id]
while True:
current_length = len(self.enc.encode(expanded_result))
if f"{source}_{left_chunk_id}" in self.chunk_id_to_index:
chunk_ids.append(left_chunk_id)
left_chunk_index = self.vectorstore.index_to_docstore_id[
self.chunk_id_to_index[f"{source}_{left_chunk_id}"]
]
left_chunk = self.vectorstore.docstore.search(left_chunk_index)
encoded_left_chunk = self.enc.encode(left_chunk.page_content)
if len(encoded_left_chunk) + current_length < target_length:
expanded_result = left_chunk.page_content + expanded_result
left_chunk_id -= 1
```

```
current_length += len(encoded_left_chunk)
else:
expanded_result += self.enc.decode(
encoded_left_chunk[-(target_length - current_length) :],
)
current_length = target_length
break
else:
left_valid = False

if f"{source}_{right_chunk_id}" in self.chunk_id_to_index:
chunk_ids.append(right_chunk_id)
right_chunk_index = self.vectorstore.index_to_docstore_id[
self.chunk_id_to_index[f"{source}_{right_chunk_id}"]
]
right_chunk = self.vectorstore.docstore.search(right_chunk_index)
encoded_right_chunk = self.enc.encode(right_chunk.page_content)
if len(encoded_right_chunk) + current_length < target_length:
expanded_result += right_chunk.page_content
right_chunk_id += 1
current_length += len(encoded_right_chunk)
else:
expanded_result += self.enc.decode(
encoded_right_chunk[: target_length - current_length],
)
current_length = target_length
break
else:
right_valid = False

if not left_valid and not right_valid:
break

expanded_chunks.append(
{
"chunk": expanded_result,
"metadata": r.metadata,
# "length": current_length,
# "chunk_ids": chunk_ids
},
)
return expanded_chunks

# Example Usage
if __name__ == "__main__":
parser = argparse.ArgumentParser(description='Retrieve documents based on a query.')
parser.add_argument('query', nargs='?', type=str, help='The query to retrieve documents for.')
args = parser.parse_args()

if not args.query:
```

```
parser.print_help()
print("Error: No query provided.")
exit(1)

# Ensure the index_folder path is correctly set in CONFIG before proceeding
index_folder = CONFIG["index_folder"]
if index_folder == "path/to/your/knowledge/directory":
print("Error: Index folder in CONFIG has not been set. Please update it to your index folder path.")
exit(1)

# Instantiate and use the DocumentRetriever with the configured index folder
retriever = DocumentRetriever(index_folder=index_folder)
query = args.query
size = 5  # Number of results to retrieve
target_length = 256  # Target length of expanded content
results = retriever(query, size, target_length)
print(results)
```

## execute_powershell_command.py

```
# Thanks to aj47:  https://github.com/aj47

import subprocess

def execute_powershell_command(command):
"""
Execute a command in PowerShell from Python.

:param command: The PowerShell command to execute as a string.
:return: The output of the command as a string.
"""
# Ensure the command is executed in PowerShell
cmd = ['powershell', '-Command', command]

# Execute the command and capture the output
try:
result = subprocess.run(cmd, capture_output=True, text=True, check=True)
return result.stdout
except subprocess.CalledProcessError as e:
return f"An error occurred: {e.stderr}"

# Example usage
if __name__ == "__main__":
command = "Get-Date"  # Example command to get the current date and time
output = execute_powershell_command(command)
print(output)
```

## fetch_web_content.py

```
#  Thanks to MADTANK:  https://github.com/madtank

from typing import Optional
```

```
import requests
import collections
collections.Callable = collections.abc.Callable
from bs4 import BeautifulSoup

def fetch_web_content(url: str) -> Optional[str]:
"""
Fetches the text content from a website.

Args:
url (str): The URL of the website.

Returns:
Optional[str]: The content of the website.
"""
try:
# Send a GET request to the URL
response = requests.get(url)

# Check for successful access to the webpage
if response.status_code == 200:
# Parse the HTML content of the page using BeautifulSoup
soup = BeautifulSoup(response.text, "html.parser")

# Extract the content of the <body> tag
body_content = soup.body

if body_content:
# Return all the text in the body tag, stripping leading/trailing whitespaces
return " ".join(body_content.get_text(strip=True).split())
else:
# Return None if the <body> tag is not found
return None
else:
# Return None if the status code isn't 200 (success)
return None
except requests.RequestException:
# Return None if any request-related exception is caught
return None
```

## generate_sd_images.py

```
# Thanks to marc-shade:  https://github.com/marc-shade
# Ollama only?  -jjg

from typing import List
import json
import requests
import io
import base64
from PIL import Image
```

```
from pathlib import Path
import uuid # Import the uuid library

# Format: protocol://server:port
base_url = "http://0.0.0.0:7860"

def generate_sd_images(query: str, image_size: str = "512x512", team_name: str = "default") -> List[str]:
"""
Function to paint, draw or illustrate images based on the users query or request.
Generates images locally with the automatic1111 API and saves them to disk.
Use the code below anytime there is a request to create an image.

:param query: A natural language description of the image to be generated.
:param image_size: The size of the image to be generated. (default is "512x512")
:param team_name: The name of the team to associate the image with.
:return: A list containing a single filename for the saved image.
"""
# Split the image size string at "x"
parts = image_size.split("x")
image_width = parts[0]
image_height = parts[1]

# list of file paths returned to AutoGen
saved_files = []

payload = {
"prompt": query,
"steps": 40,
"cfg_scale": 7,
"denoising_strength": 0.5,
"sampler_name": "DPM++ 2M Karras",
"n_iter": 1,
"batch_size": 1, # Ensure only one image is generated per batch
"override_settings": {
'sd_model_checkpoint': "starlightAnimated_v3",
}
}

api_url = f"{base_url}/sdapi/v1/txt2img"
response = requests.post(url=api_url, json=payload)

if response.status_code == 200:
r = response.json()
# Access only the final generated image (index 0)
encoded_image = r['images'][0]

image = Image.open(io.BytesIO(base64.b64decode(encoded_image.split(",", 1)[0])))

# --- Generate a unique filename with team name and UUID ---
unique_id = str(uuid.uuid4())[:8] # Get a short UUID
```

```
file_name = f"images/{team_name}_{unique_id}_output.png"

file_path = Path(file_name)
image.save(file_path)
print(f"Image saved to {file_path}")

saved_files.append(str(file_path))
else:
print(f"Failed to download the image from {api_url}")

return saved_files
```

## get_weather.py

```
import requests
from typing import Optional

def get_weather(zipcode: str, api_key: str) -> Optional[dict]:
"""
Fetches the current weather for the given ZIP code using the OpenWeatherMap API.

Args:
zipcode (str): The ZIP code for which to fetch the weather.
api_key (str): Your OpenWeatherMap API key.

Returns:
Optional[dict]: A dictionary containing the weather information, or None if an error occurs.
"""
base_url = "http://api.openweathermap.org/data/2.5/weather"
params = {
"zip": zipcode,
"appid": api_key,
"units": "imperial"  # Use "metric" for Celsius
}

try:
response = requests.get(base_url, params=params)
response.raise_for_status()  # Raise an HTTPError for bad responses
return response.json()
except requests.RequestException as e:
print(f"An error occurred: {e}")
return None

# Example usage:
# api_key = "your_openweathermap_api_key"
# weather = get_weather("94040", api_key)
# print(weather)
```

## plot_diagram.py

```
#  Thanks to MADTANK:  https://github.com/madtank
```

```
import os
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Function to draw the geometric structure with customizable file name
def draw_geometric_structure(file_name, base_circles=4, base_circle_color='blue', top_circle_color='orange',
line_color='grey', line_width=2):

# Define the directory and save path using the file_name parameter
directory = 'diagrams'
if not os.path.exists(directory):
os.makedirs(directory)
save_path = f'{directory}/{file_name}.png'

fig, ax = plt.subplots()

# Draw base circles
for i in range(base_circles):
circle = patches.Circle((i * 1.5, 0), 0.5, color=base_circle_color)
ax.add_patch(circle)

# Draw top circle
top_circle = patches.Circle(((base_circles - 1) * 0.75, 2), 0.6, color=top_circle_color)
ax.add_patch(top_circle)

# Draw lines
for i in range(base_circles):
line = plt.Line2D([(i * 1.5), ((base_circles - 1) * 0.75)], [0, 2], color=line_color, linewidth=line_width)
ax.add_line(line)

# Set limits and aspect
ax.set_xlim(-1, base_circles * 1.5)
ax.set_ylim(-1, 3)
ax.set_aspect('equal')

# Remove axes
ax.axis('off')

# Save the plot to the specified path
plt.savefig(save_path, bbox_inches='tight', pad_inches=0)
plt.close()

# Return the path for verification
return save_path

# Example usage:
file_name = 'custom_geometric_structure'
image_path = draw_geometric_structure(file_name, base_circles=8, base_circle_color='blue',
top_circle_color='orange', line_color='grey', line_width=2)
```

## save_file_to_disk.py

```
# Thanks to aj47:  https://github.com/aj47

import os

def save_file_to_disk(contents, file_name):
"""
Saves the given contents to a file with the given file name.

Parameters:
contents (str): The string contents to save to the file.
file_name (str): The name of the file, including its extension.

Returns:
str: A message indicating the success of the operation.
"""
# Ensure the directory exists; create it if it doesn't
directory = os.path.dirname(file_name)
if directory and not os.path.exists(directory):
os.makedirs(directory)

# Write the contents to the file
with open(file_name, 'w') as file:
file.write(contents)

return f"File '{file_name}' has been saved successfully."

# Example usage:
# contents_to_save = "Hello, world!"
# file_name = "example.txt"
# print(save_file_to_disk(contents_to_save, file_name))
```

## slackoverflow_teams.py

```
#  Thanks to MADTANK:  https://github.com/madtank
#  README:  https://github.com/madtank/autogenstudio-skills/blob/main/stackoverflow_teams/README.md

import os
import requests
import json
import sys

class StackOverflowTeamsSearcher:
def __init__(self):
self.api_key = os.getenv("STACK_OVERFLOW_TEAMS_API_KEY")
if not self.api_key:
raise ValueError("API key not found in environment variables")
self.base_url = "https://api.stackoverflowteams.com/2.3/search"
self.headers = {"X-API-Access-Token": self.api_key}

def search(self, query, team_name):
params = {"intitle": query, "team": team_name}
```

```
response = requests.get(self.base_url, headers=self.headers, params=params)

if response.status_code != 200:
print(f"Error: Received status code {response.status_code}")
print(response.text)
return None

try:
data = response.json()
simplified_output = []
for item in data['items']:
question = {"question": item['title']}
if 'accepted_answer_id' in item:
answer_id = item['accepted_answer_id']
answer_url = f"https://api.stackoverflowteams.com/2.3/answers/{answer_id}"
answer_params = {"team": team_name, "filter": "withbody"}
answer_response = requests.get(answer_url, headers=self.headers, params=answer_params)
if answer_response.status_code == 200:
answer_data = answer_response.json()
first_item = answer_data['items'][0]
if 'body' in first_item:
answer_text = first_item['body']
question['answer'] = answer_text
#                     else:
#                         print(f"Question {item['link']} has no answer body")
#                 else:
#                     print(f"Error: Received status code {answer_response.status_code}")
#                     print(answer_response.text)
#             else:
#                 print(f"Question {item['link']} has no answer")
simplified_output.append(question)
return json.dumps(simplified_output, indent=4)  # Pretty-printing
except ValueError as e:
print(f"Error parsing JSON: {e}")
print("Response text:", response.text)
return None

# Example Usage
if __name__ == "__main__":
if len(sys.argv) < 2:
print("Usage: python stackoverflow_teams.py <query>")
sys.exit(1)

query = sys.argv[1]
team_name = "yourteamname"  # TODO Set your team name here
# Instantiate and use the StackOverflowTeamsSearcher with the query string passed in
searcher = StackOverflowTeamsSearcher()
results = searcher.search(query, team_name)
print(results)
```

## slack_search.py

```
#  Thanks to MADTANK:  https://github.com/madtank
#  README:  https://github.com/madtank/autogenstudio-skills/blob/main/slack/README.md

import os
import requests
import json
import re
import sys

class SlackSearcher:
def __init__(self):
self.api_token = os.getenv("SLACK_API_TOKEN")
if not self.api_token:
raise ValueError("Slack API token not found in environment variables")
self.base_url = "https://slack.com/api"
self.headers = {"Authorization": f"Bearer {self.api_token}"}
# Replace these example channel names with the actual channel names you want to search
self.channel_names = ["general", "random"]

def search(self, query):
query_with_channels = self.build_query_with_channels(query)
search_url = f"{self.base_url}/search.messages"
params = {"query": query_with_channels}
response = requests.get(search_url, headers=self.headers, params=params)

if response.status_code != 200:
print(f"Error: Received status code {response.status_code}")
print(response.text)
return None

try:
data = response.json()
if not data['ok']:
print(f"Error: {data['error']}")
return None

simplified_output = []
for message in data['messages']['matches']:
simplified_message = {
"user": message['user'],
"text": message['text'],
"permalink": message['permalink']
}
thread_ts = self.extract_thread_ts(message['permalink'])
if thread_ts:
thread_messages = self.get_thread_messages(message['channel']['id'], thread_ts)
simplified_message['thread'] = thread_messages
simplified_output.append(simplified_message)
return json.dumps(simplified_output, indent=4)  # Pretty-printing
except ValueError as e:
```

```
print(f"Error parsing JSON: {e}")
print("Response text:", response.text)
return None

def build_query_with_channels(self, query):
channel_queries = [f"in:{channel}" for channel in self.channel_names]
return f"{query} {' '.join(channel_queries)}"

def extract_thread_ts(self, permalink):
match = re.search(r"thread_ts=([0-9.]+)", permalink)
return match.group(1) if match else None

def get_thread_messages(self, channel_id, thread_ts):
thread_url = f"{self.base_url}/conversations.replies"
params = {"channel": channel_id, "ts": thread_ts}
response = requests.get(thread_url, headers=self.headers, params=params)

if response.status_code != 200 or not response.json()['ok']:
print(f"Error fetching thread messages: {response.text}")
return []

thread_messages = []
for message in response.json()['messages']:
if message['ts'] != thread_ts:  # Exclude the parent message
thread_messages.append({
"user": message['user'],
"text": message['text']
})
return thread_messages

# Example Usage
if __name__ == "__main__":
if len(sys.argv) < 2:
print("Usage: python slack_search.py <query>")
sys.exit(1)

query = sys.argv[1]
searcher = SlackSearcher()
results = searcher.search(query)
print(results)
```

## webscrape.py

```
#  Thanks to MADTANK:  https://github.com/madtank

import requests
from bs4 import BeautifulSoup


def save_webpage_as_text(url, output_filename):
# Send a GET request to the URL
```

```
response = requests.get(url)

# Initialize BeautifulSoup to parse the content
soup = BeautifulSoup(response.text, 'html.parser')

# Extract text from the BeautifulSoup object
# You can adjust the elements you extract based on your needs
text = soup.get_text(separator='\n', strip=True)

# Save the extracted text to a file
with open(output_filename, 'w', encoding='utf-8') as file:
file.write(text)

# Return the file path
return output_filename



# Example usage:
# url = 'https://j.gravelle.us        /'
# output_filename = 'webpage_content.txt'
# file_path = save_webpage_as_text(url, output_filename)
# print("File saved at:", file_path)



# For a list of urls:
# urls = ['http://example.com', 'http://example.org']
# for i, url in enumerate(urls):
#     output_filename = f'webpage_content_{i}.txt'
#     save_webpage_as_text(url, output_filename)
```

## web_search.py

```
#  Thanks to MADTANK:  https://github.com/madtank
#  README:  https://github.com/madtank/autogenstudio-skills/blob/main/web_search/README.MD

import requests
from typing import List, Tuple, Optional

# Define the structure of a search result entry
ResponseEntry = Tuple[str, str, str]

# Configuration variables for the web search function
CONFIG = {
"api_provider": "google",  # or "bing"
"result_count": 3,
# For Google Search enter these values
# Refer to readme for help:  https://github.com/madtank/autogenstudio-skills/blob/main/web_search/README.MD
"google_api_key": "your_google_api_key_here",
"google_search_engine_id": "your_google_search_engine_id_here",
# Or Bing Search enter these values
"bing_api_key": "your_bing_api_key_here"
```

```
}

class WebSearch:
"""
A class that encapsulates the functionality to perform web searches using
Google Custom Search API or Bing Search API based on the provided configuration.
"""

def __init__(self, config: dict):
"""
Initializes the WebSearch class with the provided configuration.

Parameters:
- config (dict): A dictionary containing configuration settings.
"""
self.config = config

def search_query(self, query: str) -> Optional[List[ResponseEntry]]:
"""
Performs a web search based on the query and configuration.

Parameters:
- query (str): The search query string.

Returns:
- A list of ResponseEntry tuples containing the title, URL, and snippet of each result.
"""
api_provider = self.config.get("api_provider", "google")
result_count = int(self.config.get("result_count", 3))
try:
if api_provider == "google":
return self._search_google(query, cnt=result_count)
elif api_provider == "bing":
return self._search_bing(query, cnt=result_count)
except ValueError as e:
print(f"An error occurred: {e}")
except Exception as e:
print(f"An unexpected error occurred: {e}")
return None

def _search_google(self, query: str, cnt: int) -> Optional[List[ResponseEntry]]:
"""
Performs a Google search and processes the results.
Parameters:
- query (str): The search query string.
- cnt (int): The number of search results to return.

Returns:
- A list of ResponseEntry tuples containing the title, URL, and snippet of each Google search result.
"""
```

```
api_key = self.config.get("google_api_key")
search_engine_id = self.config.get("google_search_engine_id")
url = f"https://www.googleapis.com/customsearch/v1?key={api_key}&cx={search_engine_id}&q={query}"
if cnt > 0:
url += f"&num={cnt}"
response = requests.get(url)
if response.status_code == 200:
result_list: List[ResponseEntry] = []
for item in response.json().get("items", []):
result_list.append((item["title"], item["link"], item["snippet"]))
return result_list
else:
print(f"Error with Google Custom Search API: {response.status_code}")
return None

def _search_bing(self, query: str, cnt: int) -> Optional[List[ResponseEntry]]:
"""
Performs a Bing search and processes the results.

Parameters:
- query (str): The search query string.
- cnt (int): The number of search results to return.

Returns:
- A list of ResponseEntry tuples containing the name, URL, and snippet of each Bing search result.
"""
api_key = self.config.get("bing_api_key")
url = f"https://api.bing.microsoft.com/v7.0/search?q={query}"
if cnt > 0:
url += f"&count={cnt}"
headers = {"Ocp-Apim-Subscription-Key": api_key}
response = requests.get(url, headers=headers)
if response.status_code == 200:
result_list: List[ResponseEntry] = []
for item in response.json().get("webPages", {}).get("value", []):
result_list.append((item["name"], item["url"], item["snippet"]))
return result_list
else:
print(f"Error with Bing Search API: {response.status_code}")
return None

# Remember to replace the placeholders in CONFIG with your actual API keys.
# Example usage
# search = WebSearch(CONFIG)
# results = search.search_query("Example Query")
# if results is not None:
#     for title, link, snippet in results:
#         print(title, link, snippet)
```