agent_management.py

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

from api_utils import send_request_to_groq_api
from bs4 import BeautifulSoup
from ui_utils import get_api_key, 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):
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:]}."
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("No agents have yet been created. Please enter a new request.")


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}"):
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):
new_name = st.text_input("Name", value=agent['config'].get('name', ''), key=f"name_{edit_index}")
description_value = agent.get('new_description', agent.get('description', ''))
new_description = st.text_area("Description", value=description_value, key=f"desc_{edit_index}")

col1, col2, col3 = st.columns([1, 1, 2])

with col1:
if st.button(" Regenerate", 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.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)
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']
# Update the agent data in the session state
st.session_state.agents[edit_index] = agent
st.success("Agent properties updated")

# Regenerate the JSON files and zip file
regenerate_json_files_and_zip()

with col3:
# Initialize the enable_reading_html property for the agent if it doesn't exist
if "enable_reading_html" not in agent:
agent["enable_reading_html"] = False

# Use the value from the agent's enable_reading_html property for the checkbox
enable_reading_html = st.checkbox(
"Add URL reading skill to this Autogen agent",
value=agent["enable_reading_html"],
key=f"enable_reading_html_{edit_index}"
)

# Update the agent's enable_reading_html property based on the checkbox value
if enable_reading_html != agent["enable_reading_html"]:
agent["enable_reading_html"] = enable_reading_html
st.session_state.agents[edit_index] = agent  # Update the agent in the session state



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', '')
request = construct_request(agent_name, description, user_request, user_input, rephrased_request, reference_url)
response = send_request(agent_name, request)
if response:
update_discussion_and_whiteboard(agent_name, response, 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

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:
try:
response = requests.get(reference_url)
response.raise_for_status()
soup = BeautifulSoup(response.text, 'html.parser')
url_content = soup.get_text()
request += f" Reference URL content: {url_content}."
except requests.exceptions.RequestException as e:
print(f"Error occurred while retrieving content from {reference_url}: {e}")
if st.session_state.discussion:
request += f" The discussion so far has been {st.session_state.discussion[-50000:]}."

api_key = get_api_key()
if api_key is None:
st.error("API key not found. Please enter your API key.")
return

response = send_request_to_groq_api(agent_name, request, api_key)
```

```
if response:
update_discussion_and_whiteboard(agent_name, response, user_input)
# Additionally, populate the sidebar form with the agent's information
st.session_state['form_agent_name'] = agent_name
st.session_state['form_agent_description'] = description
st.session_state['selected_agent_index'] = agent_index # Keep track of the selected agent for potential
updates/deletes


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, without any additional
commentary or narrative.  It is imperative that you return ONLY the text of the new description.  No preamble, no
narrative, no superfluous commentary whatsoever.  Just the description, unlabeled, please.
"""

api_key = get_api_key()
if api_key is None:
st.error("API key not found. Please enter your API key.")
return None

print(f"regenerate_agent_description called with agent_name: {agent_name}")
print(f"regenerate_agent_description called with prompt: {prompt}")

response = send_request_to_groq_api(agent_name, prompt, api_key)
if response:
return response.strip()
else:
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):
api_key = get_api_key()
if api_key is None:
st.error("API key not found. Please enter your API key.")
return None
response = send_request_to_groq_api(agent_name, request, api_key)
return response
```

api_utils.py

```
import requests
import streamlit as st
import time

from config import RETRY_TOKEN_LIMIT


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:
raise ValueError("GROQ_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 'Regenerate' 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_to_groq_api(expert_name, request, api_key):
temperature_value = st.session_state.get('temperature', 0.1)
if api_key is None:
if 'api_key' in st.session_state and st.session_state.api_key:
api_key = st.session_state.api_key
```

```
else:
st.error("API key not found. Please enter your API key.")
return None

url = "https://api.groq.com/openai/v1/chat/completions"
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": "You are a chatbot capable of anything and everything."
},
{
"role": "user",
"content": request
}
]
}
headers = {
"Authorization": f"Bearer {api_key}",
"Content-Type": "application/json"
}

try:
response = make_api_request(url, data, headers, api_key)
if response:
if "choices" in response and len(response["choices"]) > 0:
message_content = response["choices"][0]["message"]["content"]
return message_content
else:
print("Error: Unexpected response format from the Groq API.")
print("Response data:", response)
return None
except Exception as e:
print(f"Error occurred while making the request to Groq API: {str(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
```

config.py

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

# Model configurations
MODEL_TOKEN_LIMITS = {
'llama3-70b-8192': 8192,
'llama3-8b-8192': 8192,
'mixtral-8x7b-32768': 32768,
'gemma-7b-it': 8192
}
```

file_utils.py

```
import datetime
import json
import re

def create_agent_data(agent):
expert_name = agent['config']['name']
description = agent['description']  # Use the updated description from the session state
skills = agent.get("skills", [])
# if 'enable_reading_html' is true, add skill called 'fetch_web_content' to the agent's skills
if agent.get('enable_reading_html', False):
#if skills doesn't already contain a 'fetch_web_content' entry, add it
if 'fetch_web_content' not in skills:
skills.append('fetch_web_content')

tools = agent.get("tools", [])

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

# Sanitize the description
sanitized_description = sanitize_text(description)

# Sanitize the skills and tools
sanitized_skills = [sanitize_text(skill) for skill in skills]
```

```
sanitized_tools = [sanitize_text(tool) for tool in tools]

# Create the agent data
agent_data = {
"type": "assistant",
"config": {
"name": expert_name,
"llm_config": {
"config_list": [
{
"model": "gpt-4"
}
],
"temperature": 0.1,
"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 {sanitized_description}."
},
"description": description,
"skills": sanitized_skills,
"tools": sanitized_tools
}

crewai_agent_data = {
"name": expert_name,
"description": description,
"skills": sanitized_skills,
"tools": sanitized_tools,
"verbose": True,
"allow_delegation": True
}

return 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 streamlit as st

from config import MODEL_TOKEN_LIMITS

from agent_management import display_agents
from ui_utils import get_api_key, display_api_key_input, display_discussion_and_whiteboard,
display_download_button, display_user_input, display_rephrased_request, display_reset_and_upload_buttons,
display_user_request_input, rephrase_prompt, get_agents_from_text, extract_code_from_response,
get_workflow_from_agents


def main():
st.markdown("""
<style>
/* General styles */
```

```
body {
font-family: Arial, sans-serif;
background-color: #f0f0f0;
}

/* Sidebar styles */
.sidebar .sidebar-content {
background-color: #ffffff !important;
padding: 20px !important;
border-radius: 5px !important;
box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1) !important;
}

.sidebar .st-emotion-cache-k7vsyb h1 {
font-size: 12px !important;
font-weight: bold !important;
color: #007bff !important;
}

.sidebar h2 {
font-size: 16px !important;
color: #666666 !important;
}

.sidebar .stButton button {
display: block !important;
width: 100% !important;
padding: 10px !important;
background-color: #007bff !important;
color: #ffffff !important;
text-align: center !important;
text-decoration: none !important;
border-radius: 5px !important;
transition: background-color 0.3s !important;
}

.sidebar .stButton button:hover {
background-color: #0056b3 !important;
}

.sidebar a {
display: block !important;
color: #007bff !important;
text-decoration: none !important;
}

.sidebar a:hover {
text-decoration: underline !important;
}
```

```
/* Main content styles */
.main .stTextInput input {
width: 100% !important;
padding: 10px !important;
border: 1px solid #cccccc !important;
border-radius: 5px !important;
}

.main .stTextArea textarea {
width: 100% !important;
padding: 10px !important;
border: 1px solid #cccccc !important;
border-radius: 5px !important;
resize: none !important;
}

.main .stButton button {
padding: 10px 20px !important;
background-color: #dc3545 !important;
color: #ffffff !important;
border: none !important;
border-radius: 5px !important;
cursor: pointer !important;
transition: background-color 0.3s !important;
}

.main .stButton button:hover {
background-color: #c82333 !important;
}

.main h1 {
font-size: 32px !important;
font-weight: bold !important;
color: #007bff !important;
}

/* Model selection styles */
.main .stSelectbox select {
width: 100% !important;
padding: 10px !important;
border: 1px solid #cccccc !important;
border-radius: 5px !important;
}

/* Error message styles */
.main .stAlert {
color: #dc3545 !important;
}
</style>
""", unsafe_allow_html=True)
```

```
model_token_limits = {
'llama3-70b-8192': 8192,
'llama3-8b-8192': 8192,
'mixtral-8x7b-32768': 32768,
'gemma-7b-it': 8192
}

api_key = get_api_key()
if api_key is None:
api_key = display_api_key_input()
if api_key is None:
st.warning("Please enter your GROQ_API_KEY to use the app.")
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'
)

st.title("AutoGroq")

# Ensure default values for session state are set
if "discussion" not in st.session_state:
st.session_state.discussion = ""
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)
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)

display_download_button()

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

ui_utils.py

```
import importlib.resources as resources
import os
import streamlit as st

from config import MAX_RETRIES, RETRY_DELAY
from skills.fetch_web_content import fetch_web_content

def get_api_key():
if 'api_key' in st.session_state and st.session_state.api_key:
api_key = st.session_state.api_key
print(f"API Key from session state: {api_key}")
return api_key
elif "GROQ_API_KEY" in os.environ:
api_key = os.environ["GROQ_API_KEY"]
print(f"API Key from environment variable: {api_key}")
return api_key
else:
return None



def display_api_key_input():
if 'api_key' not in st.session_state:
st.session_state.api_key = ''

api_key = st.text_input("Enter your GROQ_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 file_utils import create_agent_data, create_skill_data, sanitize_text

import datetime
import requests


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 = st.tabs(["Most Recent Comment", "Whiteboard", "Discussion History"])
with tab1:
st.text_area("Most Recent Comment", value=st.session_state.get("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)


def display_discussion_modal():
discussion_history = get_discussion_history()

with st.expander("Discussion History"):
st.write(discussion_history)


def display_download_button():
if "autogen_zip_buffer" in st.session_state and "crewai_zip_buffer" in st.session_state:
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
)
else:
st.warning("No files available for download.")


def display_user_input():
user_input = st.text_area("Additional Input:", key="user_input", height=100)
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 = {}
return user_input


def display_rephrased_request():
if "rephrased_request" not in st.session_state:
st.session_state.rephrased_request = ""

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


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():
user_request = st.text_input("Enter your request:", key="user_request", value=st.session_state.get("user_request",
""))
if st.session_state.get("previous_user_request") != user_request:
st.session_state.previous_user_request = user_request
if user_request:
if not st.session_state.get('rephrased_request'):
handle_user_request(st.session_state)
else:
autogen_agents, crewai_agents = get_agents_from_text(st.session_state.rephrased_request)
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
```

```
agents_data = {}
for agent in autogen_agents:
agent_name = agent['config']['name']
agents_data[agent_name] = agent

print(f"Debug: Agents data: {agents_data}")

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(agents_data, workflow_data, crewai_agents)
st.session_state.autogen_zip_buffer = autogen_zip_buffer
st.session_state.crewai_zip_buffer = crewai_zip_buffer
st.session_state.agents = autogen_agents

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 = []
start_index = json_string.find("{")
while start_index != -1:
end_index = json_string.find("}", start_index)
if end_index != -1:
object_str = json_string[start_index:end_index+1]
objects.append(object_str)
start_index = json_string.find("{", end_index + 1)
else:
break
```

return objects

def get_agents_from_text(text, max_retries=MAX_RETRIES, retry_delay=RETRY_DELAY):
api_key = get_api_key()
temperature_value = st.session_state.get('temperature', 0.5)
url = "https://api.groq.com/openai/v1/chat/completions"
headers = {
"Authorization": f"Bearer {api_key}",
"Content-Type": "application/json"
}
groq_request = {
"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 identify and recommend the optimal team of experts
required to fulfill this specific user's request: $userRequest Your analysis shall
consider the complexity, domain, and specific needs of the request to assemble
a multidisciplinary team of experts. Each recommended expert shall come with a defined role,
a brief description of their expertise, their skill set, and the tools they would utilize
to achieve the user's goal. The first agent must be qualified to manage the entire project,
aggregate the work done by all the other agents, and produce a robust, complete,
and reliable solution. Return the results in JSON values labeled as expert_name, description,
skills, and tools. Their 'expert_name' is their title, not their given name.
Skills and tools are arrays (one expert can have multiple skills and use multiple tools).
Return ONLY this JSON response, with no other narrative, commentary, synopsis,
or superfluous remarks/text of any kind. Tools shall be single-purpose methods,
very specific and narrow in their scope, and not at all ambiguous (e.g.: 'add_numbers'
would be good, but simply 'do_math' would be bad) Skills and tools shall be all lower case
with underscores instead of spaces, and they shall be named per their functionality,
e.g.: calculate_surface_area, or search_web
"""
},
{
"role": "user",
"content": text
}
]
}
retry_count = 0
while retry_count < max_retries:
try:
response = requests.post(url, json=groq_request, headers=headers)
if response.status_code == 200:

```
response_data = response.json()
if "choices" in response_data and response_data["choices"]:
content = response_data["choices"][0]["message"]["content"]
print(f"Content: {content}")
json_objects = extract_json_objects(content)
if json_objects:
autogen_agents = []
crewai_agents = []

missing_names = False
for json_str in json_objects:
try:
agent_data = json.loads(json_str)
expert_name = agent_data.get('expert_name', '')
if not expert_name:
missing_names = True
break
description = agent_data.get('description', '')
skills = agent_data.get('skills', [])
tools = agent_data.get('tools', [])

# Create the agent data using the new signature
autogen_agent_data = {
"type": "assistant",
"config": {
"name": expert_name,
"llm_config": {
"config_list": [
{
"model": "gpt-4"
}
],
"temperature": 0.1,
"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": skills,
"tools": tools
}

crewai_agent_data = {
"name": expert_name,
"description": description,
"skills": skills,
"tools": tools,
```

```
"verbose": True,
"allow_delegation": True
}

autogen_agents.append(autogen_agent_data)
crewai_agents.append(crewai_agent_data)
except json.JSONDecodeError as e:
print(f"Error parsing JSON object: {e}")
print(f"JSON string: {json_str}")

if missing_names:
print("Missing agent names. Retrying...")
retry_count += 1
time.sleep(retry_delay)
continue

print(f"AutoGen Agents: {autogen_agents}")
print(f"CrewAI Agents: {crewai_agents}")
return autogen_agents, crewai_agents
else:
print("No valid JSON objects found in the response")
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": None
},
"receiver": {
"type": "groupchat",
"config": {
"name": "group_chat_manager",
"llm_config": {
"config_list": [
{
"model": "gpt-4"
}
],
"temperature": temperature_value,
"cache_seed": 42,
"timeout": 600,
"max_tokens": None,
"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": None
},
"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:]]
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": [
{
"model": "gpt-4"
}
],
"temperature": temperature_value,
"cache_seed": 42,
"timeout": 600,
"max_tokens": None,
"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": None  # Set skills to null only in the workflow JSON
}

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

crewai_agents = []
for agent in agents:
_, crewai_agent_data = create_agent_data(agent)
crewai_agents.append(crewai_agent_data)

return workflow, crewai_agents


def handle_user_request(session_state):
user_request = session_state.user_request
max_retries = MAX_RETRIES
retry_delay = RETRY_DELAY

for retry in range(max_retries):
try:
rephrased_text = rephrase_prompt(user_request)
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

rephrased_text = session_state.rephrased_request

autogen_agents, crewai_agents = get_agents_from_text(rephrased_text)
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 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 rephrase_prompt(user_request):
temperature_value = st.session_state.get('temperature', 0.1)
print("Executing rephrase_prompt()")
api_key = get_api_key()
if not api_key:
st.error("API key not found. Please enter your API key.")
return None

url = "https://api.groq.com/openai/v1/chat/completions"
refactoring_prompt = f"""
Refactor the following user request into an optimized prompt for an LLM,
focusing on clarity, conciseness, and effectiveness. Provide specific details
and examples where relevant. Do NOT reply with a direct response to the request;
instead, rephrase the 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.
\n\nUser request: \"{user_request}\"\n\nrephrased:
"""

groq_request = {
"model": st.session_state.model,
"temperature": temperature_value,
"max_tokens": 100,
```

```
"top_p": 1,
"stop": "TERMINATE",
"messages": [
{
"role": "user",
"content": refactoring_prompt,
},
],
}

headers = {
"Authorization": f"Bearer {api_key}",
"Content-Type": "application/json",
}

print(f"Request URL: {url}")
print(f"Request Headers: {headers}")
print(f"Request Payload: {json.dumps(groq_request, indent=2)}")

try:
print("Sending request to Groq API...")
response = requests.post(url, json=groq_request, headers=headers, timeout=10)
print(f"Response received. Status Code: {response.status_code}")

if response.status_code == 200:
print("Request successful. Parsing response...")
response_data = response.json()
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 requests.exceptions.RequestException as e:
print(f"Error occurred while sending the request: {str(e)}")
return None
except (KeyError, ValueError) as e:
print(f"Error occurred while parsing the response: {str(e)}")
print(f"Response Content: {response.text}")
return None
except Exception as e:
print(f"An unexpected 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
response_text = f"{agent_name}:\n\n {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):
# Create separate ZIP buffers for Autogen and CrewAI
autogen_zip_buffer = io.BytesIO()
crewai_zip_buffer = io.BytesIO()
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"
agent_data = agent.copy()
agent_data['config']['name'] = formatted_agent_name
agent_file_data = json.dumps(agent_data, indent=2).encode('utf-8')  # Encode to bytes
autogen_file_data[f"agents/{agent_file_name}"] = agent_file_data
if agent.get('enable_reading_html', False):
# add skills/fetch_web_content.py to zip file
fetch_web_content_data = resources.read_text('skills', 'fetch_web_content.py')
skill_data = json.dumps(create_skill_data(fetch_web_content_data), indent=2).encode('utf-8')  # Encode to bytes
autogen_file_data[f"skills/fetch_web_content.json"] = skill_data

workflow_file_name = "workflow.json"
workflow_file_data = json.dumps(workflow_data, indent=2).encode('utf-8')  # Encode to bytes
autogen_file_data[workflow_file_name] = workflow_file_data

# Prepare CrewAI 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 = json.dumps(crewai_agent_data, indent=2).encode('utf-8')  # Encode to bytes
crewai_file_data[f"agents/{agent_file_name}"] = agent_file_data

# Create ZIP files
create_zip_file(autogen_zip_buffer, autogen_file_data)
create_zip_file(crewai_zip_buffer, crewai_file_data)
```

```
# Move the ZIP file pointers to the beginning
autogen_zip_buffer.seek(0)
crewai_zip_buffer.seek(0)

return autogen_zip_buffer, crewai_zip_buffer
```

fetch_web_content.py

```
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
```