agent_management.py

```python
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,
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)
```

```python
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}",
                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()
            space = container.empty()
            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
```

```python
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', ''))
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("Re-roll ", 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']
st.session_state.agents[edit_index] = agent
regenerate_json_files_and_zip()
st.session_state['show_edit'] = False
with col3:
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_file in skill_files:
skill_name = os.path.splitext(skill_file)[0]
if skill_name not in agent:
agent[skill_name] = False

skill_checkbox = st.checkbox(
f"Add {skill_name} skill to this agent in Autogen™",
```

```python
                value=agent[skill_name],
                key=f"{skill_name}_{edit_index}"
            )

            if skill_checkbox != agent[skill_name]:
                agent[skill_name] = skill_checkbox
                st.session_state.agents[edit_index] = agent


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}")


# agent_management.py

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:
```

```python
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)
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']
```

```python
    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)
```

```python
    return response
```

api_utils.py

```python
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 'Re-roll' 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",
```

```python
    "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

```python
# 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

```python
import datetime
import importlib.resources as resources
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)

autogen_agent_data = {
"type": "assistant",
"config": {
"name": formatted_expert_name,
"llm_config": {
"config_list": [
{
"user_id": "default",
"timestamp": current_timestamp,
"model": "gpt-4",
"base_url": None,
"api_type": None,
"api_version": None,
"description": "OpenAI model configuration"
}
],
"temperature": st.session_state.get('temperature', 0.1),
"cache_seed": None,
"timeout": None,
"max_tokens": None,
```

```python
            "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()
```

```python
    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

```python
import os
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, load_skill_functions


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

```python
# 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:
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)
```

```python
        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

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

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")
```

```python
        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(
```

```python
        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)
    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_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,
```

```python
        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:
```

```python
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(workflow_data)
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("{")
```

```python
    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 AI agents 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. 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.

Guidelines:
1. **Project Manager**: The first agent must be qualified to manage the entire project, aggregate the work done by all other agents, and produce a robust, complete, and reliable solution.
2. **Agent Roles**: Clearly define each agent's role in the project.
3. **Expertise Description**: Provide a brief but thorough description of each agent's expertise.
4. **Specific Skills**: List the specific skills of each agent.
5. **Specific Tools**: List the specific tools each agent would utilize. Tools must be single-purpose methods, very specific, and not ambiguous (e.g., 'add_numbers' is good, but 'do_math' is bad).
6. **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).
7. **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).
8. **Execution Focus**: Agents should focus on executing tasks and providing actionable steps rather than just

planning. They should break down tasks into specific, executable actions and delegate subtasks to other agents or utilize their skills when appropriate.
9. **Step-by-Step Solutions**: Agents should move from the planning phase to the execution phase as quickly as possible and provide step-by-step solutions to the user's request.

Return ONLY the JSON response, with no other narrative, commentary, synopsis, or superfluous text of any kind.
"""
    },
    {
    "role": "user",
    "content": text
    }
    ]
    }

```python
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', [])
                            # Associate skills with the agent based on their capabilities
                            agent_skills = []
                            for skill_name in skills:
                                if skill_name in st.session_state.skill_functions:
                                    agent_skills.append(skill_name)
                            # Create the agent data using the new signature
                            autogen_agent_data = {
                                "type": "assistant",
                                "config": {
                                    "name": expert_name,
                                    "llm_config": {
```

```python
"config_list": [
{
"user_id": "default",
"timestamp": datetime.datetime.now().isoformat(),
"model": "gpt-4",
"base_url": None,
"api_type": None,
"api_version": None,
"description": "OpenAI model configuration"
}
],
"temperature": st.session_state.get('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": 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)
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")
```

```python
        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": "gpt-4",
    "base_url": None,
    "api_type": None,
    "api_version": None,
    "description": "OpenAI model configuration"
    }
    ],
    "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": []
    },
    "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
```

```python
    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": "gpt-4",
"base_url": None,
"api_type": None,
"api_version": None,
"description": "OpenAI model configuration"
}
],
"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": []  # Set skills to null only in the workflow JSON
}

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

```python
        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):
    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
```

```python
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 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 a language model. Focus on the following aspects:
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 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.

User request: "{user_request}"

Rephrased:
"""

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

```python
                    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):
    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"
        autogen_agent_data, _ = create_agent_data(agent)
        autogen_agent_data['config']['name'] = formatted_agent_name
        agent_file_data = json.dumps(autogen_agent_data, indent=2).encode('utf-8')
        autogen_file_data[f"agents/{agent_file_name}"] = agent_file_data

    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")]
```

```python
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 = json.dumps(create_skill_data(skill_data), indent=2).encode('utf-8')
autogen_file_data[f"skills/{skill_name}.json"] = skill_json

workflow_file_name = "workflow.json"
workflow_file_data = json.dumps(workflow_data, indent=2).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 = json.dumps(crewai_agent_data, indent=2).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
```

fetch_post.py

```python
# Note to AutoGen, please follow these instructions to run the skill properly
# Example of fetching messages using the fetch_post skill:
# from skills import fetch_post CONFIG
# response = fetch_post(action='fetch')
# print(response)

# # Ensure that you replace the above 'fetch' action with 'post' when you want to post a message,
# # and provide the necessary 'message' arguments.
# Example of posting messages using the fetch_post skill:
# from skills import fetch_post
# response = fetch_post(action='post', message=<message>)
# print(response)

import requests
import logging
from datetime import datetime
```

```python
# Global configuration variables
USERNAME = "AutoGen-Proxy-User"  # Change this value to your user name
LAMBDA_URL = "https://m7cjbptdpsuj56rrx7e6qhq7ou0svley.lambda-url.us-west-2.on.aws/"  # Playground Chat
TOPICS = ["autogen"]
PERSONALITY = "Technical"


def fetch_post(action='fetch', message=None, username=None):
    """
    Processes the given action, either fetching or posting a message.
    """
    global USERNAME
    username = username or USERNAME
    if action == 'fetch':
        return fetch_messages()
    elif action == 'post':
        return post_message(message, username)
    else:
        return "Invalid action specified."


def fetch_messages():
    """
    Fetches messages from the lambda URL endpoint.
    """
    global LAMBDA_URL, TOPICS, PERSONALITY
    lambda_url = LAMBDA_URL + "fetch"

    try:
        response = requests.get(lambda_url)
        if response.ok:
            raw_messages = response.json()
            formatted_messages = format_messages(raw_messages, TOPICS, PERSONALITY)
            return {"messages": formatted_messages, "system_message": system_message(TOPICS, PERSONALITY)}
        else:
            logging.error(f"Failed to fetch posts. Response: {response.text}")
            return "Failed to fetch posts."
    except Exception as e:
        logging.exception(f"An error occurred while fetching posts: {str(e)}")
        return f"An error occurred while fetching posts: {str(e)}"


def post_message(message, username):
    """
    Posts a message to the lambda URL endpoint.
    """
    global LAMBDA_URL
    lambda_url = LAMBDA_URL + "post"
    payload = {'username': username, 'message': message}

    try:
        response = requests.post(lambda_url, json=payload)
        if response.ok:
```

```python
        return f"Message from {username}: '{message}' posted successfully to Fetch Post."
    else:
        logging.error(f"Failed to post message. Response: {response.text}")
        return "Failed to post message."
except Exception as e:
    logging.exception("An error occurred while posting the message.")
    return f"An error occurred while posting the message: {str(e)}"


def format_messages(raw_messages, topics, personality):
    """
    Formats the raw messages into a readable structure based on the topics and personality.

    Parameters:
    - raw_messages (list): The list of message dictionaries to format.
    - topics (list): The list of topics to focus on.
    - personality (str): The personality setting for the messages.

    Returns:
    - A list of formatted message dictionaries.
    """
    formatted_messages = []
    for message in raw_messages:
        timestamp = datetime.fromtimestamp(message['Timestamp'])
        formatted_time = timestamp.strftime('%H:%M:%S %m/%d/%Y')
        formatted_messages.append({
            "Timestamp": formatted_time,
            "Message": message['Message'],
            "Username": message['Username'],
            "MessageID": message['MessageID']
        })
    return formatted_messages


def system_message(topics, personality):
    """
    Generates a system message for fetched posts, providing context for the AI.

    Parameters:
    - topics (list): The list of topics that the messages are about.
    - personality (str): The personality setting of the AI.

    Returns:
    - A formatted string with the system message.
    """
    return (
        "AutoGenStudio, you've fetched the latest messages from the Fetch Post. "
        "Focus on topics: " + ', '.join(topics) + ". "
        "Use this information for formulating responses, if needed. "
        "Personality setting: " + personality + "."
    )
```

```python
# Example usage of the fetch_post function
# response = fetch_post(action='fetch')
# print(response)
```

fetch_web_content.py

```python
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
```