Table of Contents

agent_management.py

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

from api_utils import send_request_to_groq_api
from file_utils import create_agent_data
from ui_utils import 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 delete_agent(index):
if 0 <= index < len(st.session_state.agents):
expert_name = st.session_state.agents[index]["expert_name"]
del st.session_state.agents[index]

# Get the full path to the JSON file
agents_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), "agents"))
json_file = os.path.join(agents_dir, f"{expert_name}.json")

# Delete the corresponding JSON file
if os.path.exists(json_file):
os.remove(json_file)
print(f"JSON file deleted: {json_file}")
else:
print(f"JSON file not found: {json_file}")

st.experimental_rerun()


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")
for index, agent in enumerate(st.session_state.agents):
agent_name = agent["config"]["name"]
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.sidebar.markdown(button_style, unsafe_allow_html=True)
st.sidebar.button(agent_name, key=f"agent_{index}", on_click=agent_button_callback(index))
else:
st.sidebar.warning("No agents created. Please enter a new request.")


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):
# Retrieve agent information using the provided index
agent = st.session_state.agents[agent_index]

# Preserve the original "Act as" functionality
agent_name = agent["config"]["name"]
description = agent["description"]
user_request = st.session_state.get('user_request', '')
user_input = st.session_state.get('user_input', '')
rephrased_request = st.session_state.get('rephrased_request', '')

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 st.session_state.discussion:
request += f" The discussion so far has been {st.session_state.discussion[-50000:]}."

response = send_request_to_groq_api(agent_name, request)
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
```

api_utils.py

```
import datetime
import requests
import json
import streamlit as st
import re
import time

from file_utils import create_agent_data, sanitize_text
from skills.stock_info_skill import GetStockInfo


def call_coordinating_agent_api(last_agent, last_comment, agents, enhanced_prompt):
expert_names = [agent["config"]["name"] for agent in agents]
return get_next_agent(last_agent, last_comment, expert_names, enhanced_prompt)


def get_next_agent(last_agent, last_comment, expert_names, enhanced_prompt):
url = "https://j.gravelle.us/APIs/Groq/groqApiChatCoordinator.php"
data = {
"last_agent": last_agent,
"last_contribution": last_comment,
"agents": expert_names, # Pass the expert names instead of the entire agent objects
"enhanced_prompt": enhanced_prompt
}
headers = {"Content-Type": "application/json"}

print("Payload:")
print(json.dumps(data, indent=2))

try:
response = requests.post(url, json=data, headers=headers)
print(f"Debug: RESPONSE: {response.text}")
response.raise_for_status()
response_data = response.json()
print(f"Debug: RESPONSE DATA: {response_data}")
next_agent = response_data["next_agent"].strip()
assignment = response_data["assignment"].strip()

if next_agent not in expert_names:
print(f"Warning: The returned next agent '{next_agent}' is not one of the provided expert names:
{expert_names}")
print("Falling back to the last agent.")
next_agent = last_agent
assignment = "Please continue working on the task based on the previous assignment and the
enhanced prompt."

return f"Next Suggested Agent: {next_agent}\n\nAssignment: {assignment}\n"
except (requests.exceptions.RequestException, KeyError, ValueError) as e:
print(f"Error occurred while coordinating agents:")
print(f"Request URL: {url}")
print(f"Request Headers: {headers}")
print(f"Request Payload: {json.dumps(data, indent=2)}")
print(f"Response Content: {response.text}")
print(f"Error Details: {str(e)}")
```

```
return "Error occurred while coordinating agents."
except Exception as e:
print(f"An unexpected error occurred:")
print(f"Error Details: {str(e)}")
return "Error occurred while coordinating agents."


def extract_tasks(comment, agents):
url = "https://j.gravelle.us/APIs/Groq/groqApiTaskExtractor.php"
data = {
"comment": comment,
"agents": agents
}
headers = {"Content-Type": "application/json"}
response = requests.post(url, json=data, headers=headers)
response.raise_for_status()
response_data = response.json()

return response_data


def make_api_request(url, data, headers):
max_retries = 3
retry_delay = 1 # in seconds

for retry in range(max_retries):
try:
time.sleep(1) # Add a 1-second delay before making the API request
response = requests.post(url, data=json.dumps(data), headers=headers)
print(f"Debug: API request sent: {json.dumps(data)}")
print(f"Debug: API response received: {response.text}")

if response.status_code == 200:
try:
return response.json()
except json.JSONDecodeError:
print(f"Error: Unexpected response format: {response.text}")
return None
else:
st.error(f"Error: API request failed with status code {response.status_code}. Retrying...")
if retry < max_retries - 1:
time.sleep(retry_delay)
continue
else:
return None
except requests.exceptions.RequestException as e:
st.error(f"Error: {str(e)}. Retrying...")
if retry < max_retries - 1:
time.sleep(retry_delay)
continue
else:
return None

return None


def rephrase_prompt(user_request):
url = "https://j.gravelle.us/APIs/Groq/groqApiRephrasePrompt.php"
data = {"user_request": user_request}
headers = {"Content-Type": "application/json"}
response_data = make_api_request(url, data, headers)
if response_data:
```

```
rephrased = response_data.get("rephrased", "")
if rephrased:
return rephrased
else:
print("Error: Empty response received from the API.")
return None



def get_agents_from_text(text):
url = "https://j.gravelle.us/APIs/Groq/groqApiGetAgentsFromPrompt.php"
data = {"user_request": text}
headers = {"Content-Type": "application/json"}
response_data = make_api_request(url, data, headers)
if response_data:
autogen_agents = []
crewai_agents = []

if isinstance(response_data, dict):
for expert_name, agent_data in response_data.items():
expert_name = agent_data.get("expert_name", "")
description = agent_data.get("description", "")
skills = agent_data.get("skills", [])
tools = agent_data.get("tools", [])
autogen_agent_data, crewai_agent_data = create_agent_data(expert_name, description, skills, tools)
autogen_agents.append(autogen_agent_data)
crewai_agents.append(crewai_agent_data)
elif isinstance(response_data, list):
for agent_data in response_data:
expert_name = agent_data.get("expert_name", "")
description = agent_data.get("description", "")
skills = agent_data.get("skills", [])
tools = agent_data.get("tools", [])
autogen_agent_data, crewai_agent_data = create_agent_data(expert_name, description, skills, tools)
autogen_agents.append(autogen_agent_data)
crewai_agents.append(crewai_agent_data)
else:
print("Error: Unexpected response format from the API.")

return autogen_agents, crewai_agents

return [], []



def get_workflow_from_agents(agents):
current_timestamp = datetime.datetime.now().isoformat()

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-4-1106-preview"
}
],
"temperature": 0.1,
"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-4-1106-preview"
}
],
"temperature": 0.1,
"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 index, agent in enumerate(agents):
agent_name = agent["config"]["name"]
description = agent["description"]
_, crewai_agent_data = create_agent_data(agent_name, description, agent.get("skills"),
agent.get("tools"))
crewai_agents.append(crewai_agent_data)

return workflow, crewai_agents


# api_utils.py
def send_request_to_groq_api(expert_name, request):
url = "https://j.gravelle.us/APIs/Groq/groqApiStockDiscerner.php"

# Extract the text that follows "Additional input:" from the request
additional_input_index = request.find("Additional input:")
if additional_input_index != -1:
additional_input = request[additional_input_index + len("Additional input:"):].strip()
else:
additional_input = ""

if additional_input:
data = {"user_request": additional_input}
headers = {"Content-Type": "application/json"}

try:
```

```
response = requests.post(url, json=data, headers=headers)
response.raise_for_status()

try:
response_data = response.json()
if "summary" in response_data:
summary = response_data["summary"].strip()
else:
summary = ""
except ValueError:
summary = response.text.strip()

if summary.startswith("LOOKUP"):
ticker = summary.split("LOOKUP")[1].strip()
stock_info = GetStockInfo(ticker)
request += f"\n\nStock info: {stock_info}"

except requests.exceptions.RequestException as e:
print(f"Error occurred while making the request: {e}")
except Exception as e:
print(f"An unexpected error occurred: {e}")

url = "https://j.gravelle.us/APIs/Groq/groqAPI.php"
data = {
"model": st.session_state.model,
"temperature": 0.5,
"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 = {"Content-Type": "application/json"}
response_data = make_api_request(url, data, headers)
if response_data:
message_content = response_data["choices"][0]["message"]["content"]
return message_content
return ""


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)
```

custom_button.py

```
import streamlit as st
import streamlit.components.v1 as components

def custom_button(expert_name, index, next_agent):
button_style = """
<style>
.custom-button {
background-color: #f0f0f0;
color: black;
padding: 0.5rem 1rem;
border: none;
border-radius: 0.25rem;
cursor: pointer;
}
.custom-button.active {
background-color: green;
color: white;
}
</style>
"""

button_class = "custom-button active" if next_agent == expert_name else "custom-button"
button_html = f'<button class="{button_class}">{expert_name}</button>'

components.html(button_style + button_html, height=50)

def agent_button(expert_name, index, next_agent):
custom_button(expert_name, index, next_agent)
```

file_utils.py

```
# file_utils.py
import os
import json
import re


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


def create_agent_data(expert_name, description, skills=None, tools=None):
# 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] if skills else []
```

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

# Create the agent data
agent_data = {
"type": "assistant",
"config": {
"name": formatted_expert_name,
"llm_config": {
"config_list": [
{
"model": "gpt-4-1106-preview"
}
],
"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": sanitized_description,
"skills": [],
"tools": sanitized_tools
}

crewai_agent_data = {
"name": expert_name, # Use 'name' instead of 'expert_name'
"description": description, # Use 'description' instead of 'goal'
"skills": skills, # Add 'skills' key
"tools": sanitized_tools,
"verbose": True,
"allow_delegation": True
}

return agent_data, crewai_agent_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
```

main.py

```
import streamlit as st
from agent_management import display_agents
from ui_utils import display_discussion_and_whiteboard, display_download_button,
display_user_input, display_rephrased_request, display_reset_button, display_user_request_input


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

col1, col2, col3 = st.columns([2, 5, 3])
with col3:
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]

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_button()
st.markdown('</div>', unsafe_allow_html=True)

display_download_button()

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

ui_utils.py

```
import io
import json
import os
import streamlit as st
import time
import zipfile
from api_utils import call_coordinating_agent_api, rephrase_prompt, get_agents_from_text,
extract_code_from_response, get_workflow_from_agents
from file_utils import create_agent_data, sanitize_text


def display_discussion_and_whiteboard():
col1, col2 = st.columns(2)
with col1:
st.text_area("Discussion", value=st.session_state.discussion, height=400, key="discussion")
with col2:
st.text_area("Whiteboard", value=st.session_state.whiteboard, height=400, key="whiteboard")

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

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


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_reset_button():
if st.button("Reset", key="reset_button"):
# Reset specific elements without clearing entire session state
for key in ["rephrased_request", "discussion", "whiteboard", "user_request", "user_input", "agents",
"zip_buffer"]:
if key in st.session_state:
del st.session_state[key]

st.session_state.user_request = ""

st.session_state.show_begin_button = True
st.experimental_rerun()



def display_user_request_input():
user_request = st.text_input("Enter your request:", key="user_request")
if user_request and user_request != st.session_state.get("previous_user_request"):
st.session_state.previous_user_request = user_request
handle_begin(st.session_state)
st.experimental_rerun()



def handle_begin(session_state):
user_request = session_state.user_request
max_retries = 3
retry_delay = 1 # in seconds

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
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. Retrying...")
if retry < max_retries - 1:
time.sleep(retry_delay)
continue
else:
print("Error: Failed to create agents after maximum retries.")
```

```
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)
session_state.autogen_zip_buffer = autogen_zip_buffer
session_state.crewai_zip_buffer = crewai_zip_buffer

session_state.agents = autogen_agents
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_begin: {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



def update_discussion_and_whiteboard(expert_name, response, user_input):
print("Updating discussion and whiteboard...")
print(f"Expert Name: {expert_name}")
print(f"Response: {response}")
print(f"User Input: {user_input}")

if user_input:
user_input_text = f"\n\nAdditional Input:\n\n{user_input}\n\n"
st.session_state.discussion += user_input_text

response_text = f"{response}\n\n===\n\n"
st.session_state.discussion += response_text

code_blocks = extract_code_from_response(response)
st.session_state.whiteboard = code_blocks

# Store the last agent and their comment in session variables
st.session_state.last_agent = expert_name
st.session_state.last_comment = response

print(f"Last Agent: {st.session_state.last_agent}")
print(f"Last Comment: {st.session_state.last_comment}")

# Check if there are at least two agents in the discussion
```

```
if len(st.session_state.agents) >= 2:
print("Sufficient agents in the discussion. Calling coordinating agent API...")
print(f"Agents: {st.session_state.agents}")
print(f"Enhanced Prompt: {st.session_state.rephrased_request}")

# Call the internal coordinating agent API
coordinating_agent_response = call_coordinating_agent_api(
st.session_state.last_agent,
st.session_state.last_comment,
st.session_state.agents,
st.session_state.rephrased_request
)
print(coordinating_agent_response)

print(f"Coordinating Agent Response: {coordinating_agent_response}")

# Append the coordinating agent's response to the discussion
st.session_state.discussion += f"\n\n{coordinating_agent_response}\n\n"
else:
print("Insufficient agents in the discussion. Skipping coordinating agent API call.")


def zip_files_in_memory(agents_data, workflow_data, crewai_agents):
# Create separate ZIP buffers for Autogen and CrewAI
autogen_zip_buffer = io.BytesIO()
crewai_zip_buffer = io.BytesIO()

# Create a ZIP file in memory
with zipfile.ZipFile(autogen_zip_buffer, 'w', zipfile.ZIP_DEFLATED) as zip_file:
# Write agent files to the ZIP
for agent_name, agent_data in agents_data.items():
agent_file_name = f"{agent_name}.json"
agent_file_data = json.dumps(agent_data, indent=2)
zip_file.writestr(f"agents/{agent_file_name}", agent_file_data)

# Write workflow file to the ZIP
workflow_file_name = f"{sanitize_text(workflow_data['name'])}.json"
workflow_file_data = json.dumps(workflow_data, indent=2)
zip_file.writestr(f"workflows/{workflow_file_name}", workflow_file_data)

with zipfile.ZipFile(crewai_zip_buffer, 'w', zipfile.ZIP_DEFLATED) as zip_file:
for index, agent_data in enumerate(crewai_agents):
agent_file_name = f"agent_{index}.json"
agent_file_data = json.dumps(agent_data, indent=2)
zip_file.writestr(f"agents/{agent_file_name}", agent_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
```

skills\stock_info_skill.py

```
import requests

def GetStockInfo(ticker):
url = f"https://j.gravelle.us/APIs/Stocks/tickerApi.php?q={ticker}"
response = requests.get(url)
if response.status_code == 200:
data = response.json()
```

```
if data["status"] == "OK" and data["resultsCount"] > 0:
result = data["results"][0]
return f"Stock info for {ticker}:\nOpen: {result['o']}\nClose: {result['c']}\nHigh: {result['h']}\nLow:
{result['l']}\nVolume: {result['v']}"
return f"No stock info found for {ticker}"
```