Pulling Google Drive Data into RAG with LangServe and Docker

Deploying a RAG toolchain using LangServe in your local Docker environment

Purpose

- Learn to leverage LangChain Templates and Docker to build scalable microservices for your chains with ease

- See how you can interact with LangServe to create FastAPI endpoints for dynamically create and invoke your chains
- Streamlit as frontend to interact with your chains
- Docker to containerize and deploy your AI microservices

- See the app in action performing RAG on your own data

- Try it out and create your own with your preferred document loader!

Use Cases for RAG on your own Data

- Retrieving engineering standards / procedures
- Querying meeting transcripts
- Getting insights on project info that would otherwise be overlooked

Lab Deep Dive

Technologies Used

Backend

- LangChain rag-chroma template
- Fast API + LangServe
- Open AI - ChatOpenAI + OpenAIEmbeddings
- Chroma
- Google Drive API

Frontend

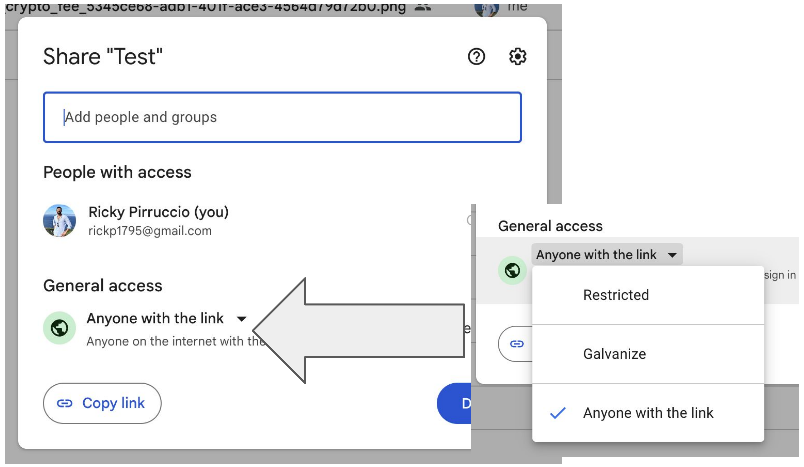
- Streamlit - link to tutorial

Deployment and Development Pipeline

- Docker - link to tutorial

Google Credentials Prerequisites

- You must create a google service account before you can access the Google Drive API on your own folders
- Karim Lalani made a great tutorial on this here
- Important: Set your folder sharing permissions to “Anyone with the link”



crypto_fee_5345ce68-adb1-401f-ace3-4564d79d72b0.png

Share "Test"

Add people and groups

People with access

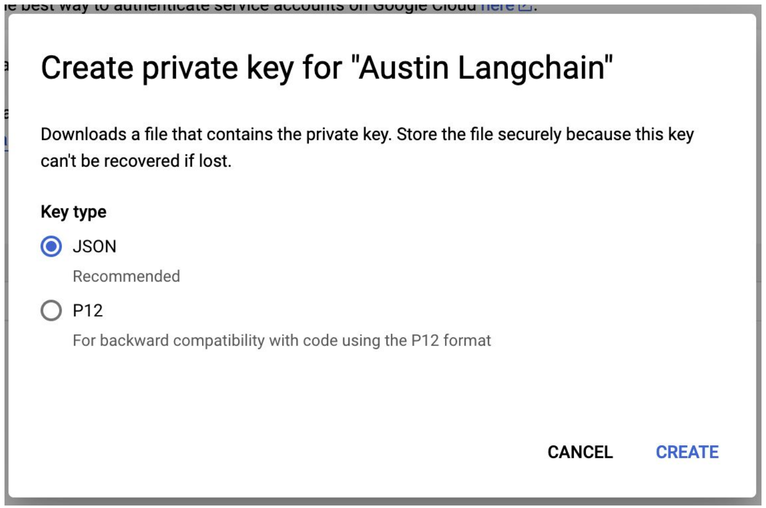
Ricky Pirruccio (you)
rickp1795@gmail.com

General access

Anyone with the link
Anyone on the internet with the

Google Credentials Prerequisites

- After creating a service account, you'll receive a key file with credentials for authentication. Rename it keys.json and save it in this path ~/.credentials/keys.json



Create private key for "Austin Langchain"

Downloads a file that contains the private key. Store the file securely because this key can't be recovered if lost.

Key type

[✓ JSON]
Recommended

P12
For backward compatibility with code using the P12 format

Prerequisites

Clone the repo

- git clone https://github.com/colinmcnamara/austin_langchain
- cd austin_langchain/labs/LangChain_103/rag_chroma_from_google_drive

Installs

- [Docker Desktop]()

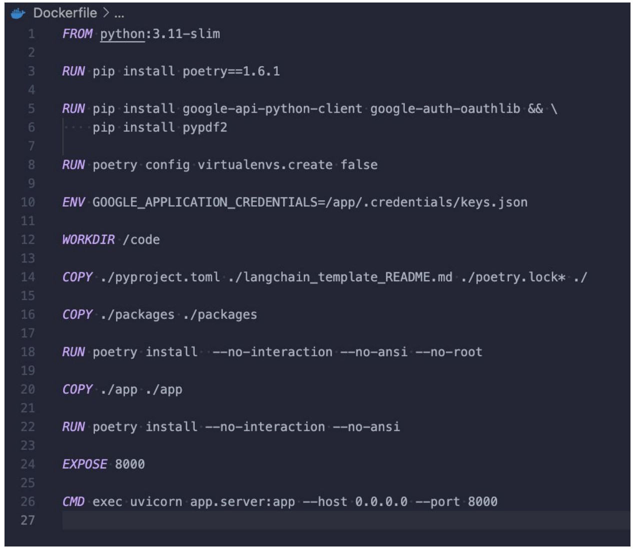
Environment Variables

- `GOOGLE_APPLICATION_CREDENTIALS:` Set this variable to the path where you saved the keys.json file earlier. Make sure to call out the full path or you will get an error
- `OPENAI_API_KEY`

Live Demo

Docker setup

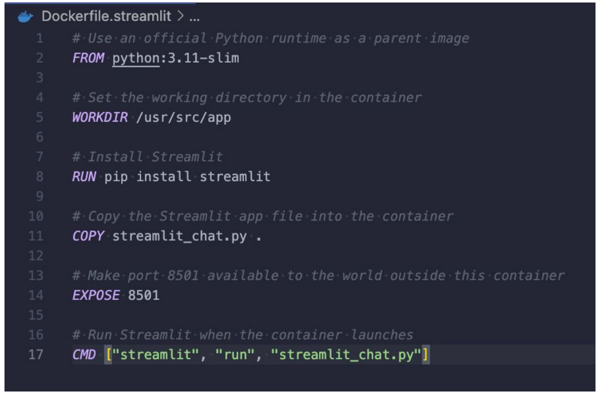
Chain Microservice (Backend)



Dockerfile > ...

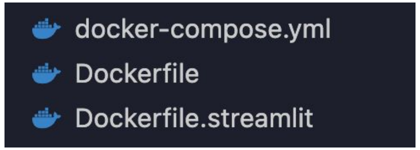
```dockerfile
FROM python:3.11-slim

RUN pip install poetry==1.6.1

RUN pip install google-api-python-client google-auth-oauthlib && \
    pip install pypdf2

RUN poetry config virtualenvs.create false

ENV GOOGLE_APPLICATION_CREDENTIALS=/app/.credentials/keys.json

WORKDIR /code

COPY ./pyproject.toml ./langchain_template_README.md ./poetry.lock* ./

COPY ./packages ./packages

RUN poetry install  --no-interaction --no-ansi --no-root

COPY ./app ./app

RUN poetry install --no-interaction --no-ansi

EXPOSE 8000

CMD exec uvicorn app.server:app --host 0.0.0.0 --port 8000
```

Streamlit Microservice (Frontend)



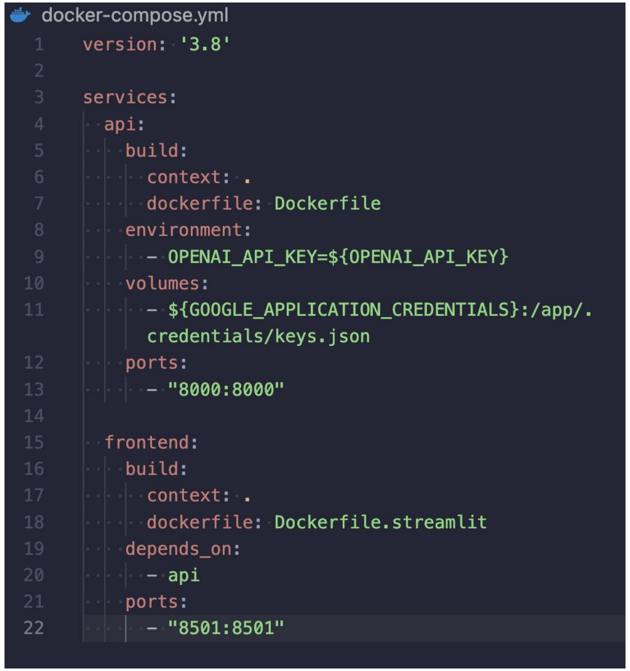
Dockerfile.streamlit > ...

```dockerfile
# Use an official Python runtime as a parent image
FROM python:3.11-slim

# Set the working directory in the container
WORKDIR /usr/src/app

# Install Streamlit
RUN pip install streamlit

# Copy the Streamlit app file into the container
COPY streamlit_chat.py .

# Make port 8501 available to the world outside this container
EXPOSE 8501

# Run Streamlit when the container launches
CMD ["streamlit", "run", "streamlit_chat.py"]
```



docker-compose.yml
Dockerfile
Dockerfile.streamlit

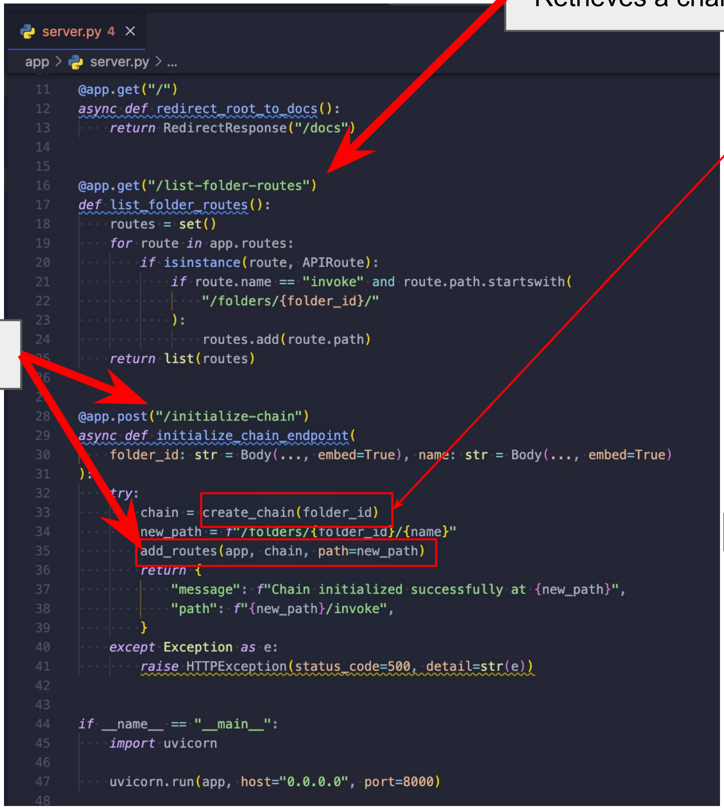
Compose



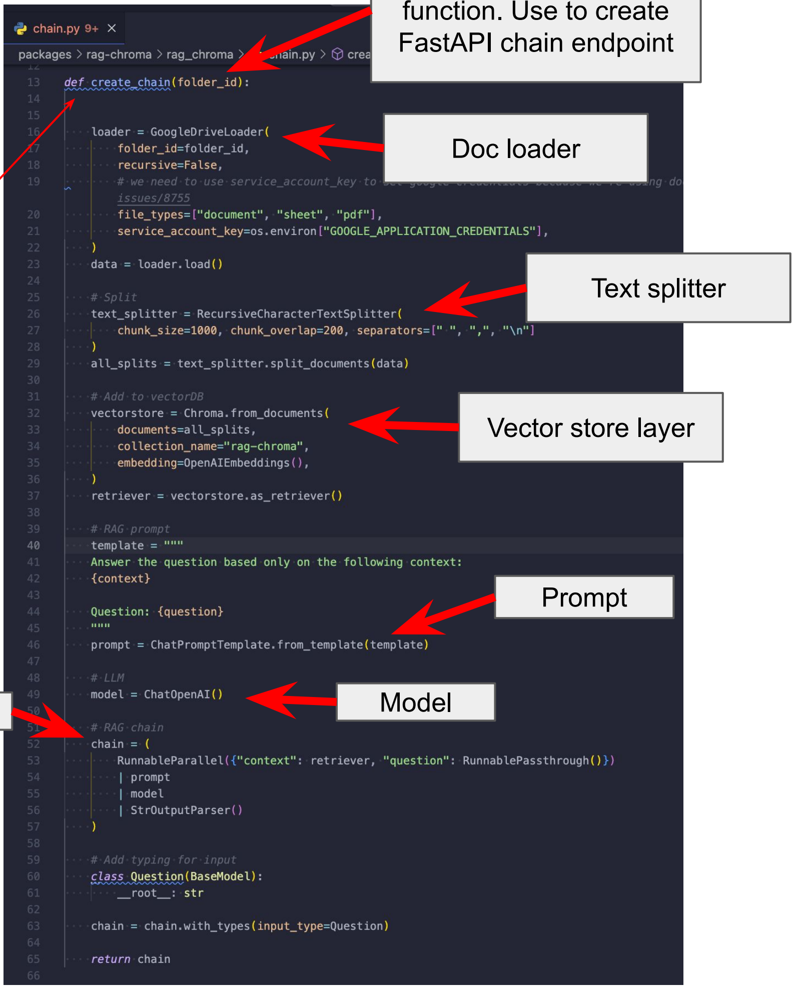
docker-compose.yml

```yaml
version: '3.8'

services:
  api:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - OPENAI_API_KEY=${OPENAI_API_KEY}
    volumes:
      - ${GOOGLE_APPLICATION_CREDENTIALS}:/app/.credentials/keys.json
    ports:
      - "8000:8000"

  frontend:
    build:
      context: .
      dockerfile: Dockerfile.streamlit
    depends_on:
      - api
    ports:
      - "8501:8501"
```

Basic Backend Setup



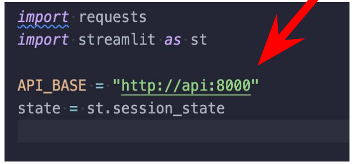
```python
@app.get("/")
async def redirect_root_to_docs():
    return RedirectResponse("/docs")


@app.get("/list-folder-routes")
def list_folder_routes():
    routes = set()
    for route in app.routes:
        if isinstance(route, APIRoute):
            if route.name == "invoke" and route.path.startswith(
                "/folders/{folder_id}/"
            ):
                routes.add(route.path)
    return list(routes)


@app.post("/initialize-chain")
async def initialize_chain_endpoint(
    folder_id: str = Body(..., embed=True), name: str = Body(..., embed=True)
):
    try:
        chain = create_chain(folder_id)
        new_path = f"/folders/{folder_id}/{name}"
        add_routes(app, chain, path=new_path)
        return {
            "message": f"Chain initialized successfully at {new_path}",
            "path": f"{new_path}/invoke",
        }
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))


if __name__ == "__main__":
    import uvicorn

    uvicorn.run(app, host="0.0.0.0", port=8000)
```


chain.py 9+
packages > rag-chroma > rag_chroma > chain.py > crea
rap chain creation in a function. Use to create FastAPI chain endpoint
Doc loader
Text splitter
Vector store layer
Prompt
Model
Chain

```python
def create_chain(folder_id):


    loader = GoogleDriveLoader(
        folder_id=folder_id,
        recursive=False,
        # we need to use service_account_key to set google credentials because we're using do
        issues/8755
        file_types=["document", "sheet", "pdf"],
        service_account_key=os.environ["GOOGLE_APPLICATION_CREDENTIALS"],
    )
    data = loader.load()

    # Split
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000, chunk_overlap=200, separators=[" ", ",", "\n"]
    )
    all_splits = text_splitter.split_documents(data)

    # Add to vectorDB
    vectorstore = Chroma.from_documents(
        documents=all_splits,
        collection_name="rag-chroma",
        embedding=OpenAIEmbeddings(),
    )
    retriever = vectorstore.as_retriever()

    # RAG prompt
    template = """
    Answer the question based only on the following context:
    {context}

    Question: {question}
    """
    prompt = ChatPromptTemplate.from_template(template)

    # LLM
    model = ChatOpenAI()

    # RAG chain
    chain = (
        RunnableParallel({"context": retriever, "question": RunnablePassthrough()})
        | prompt
        | model
        | StrOutputParser()
    )

    # Add typing for input
    class Question(BaseModel):
        __root__: str

    chain = chain.with_types(input_type=Question)

    return chain
```

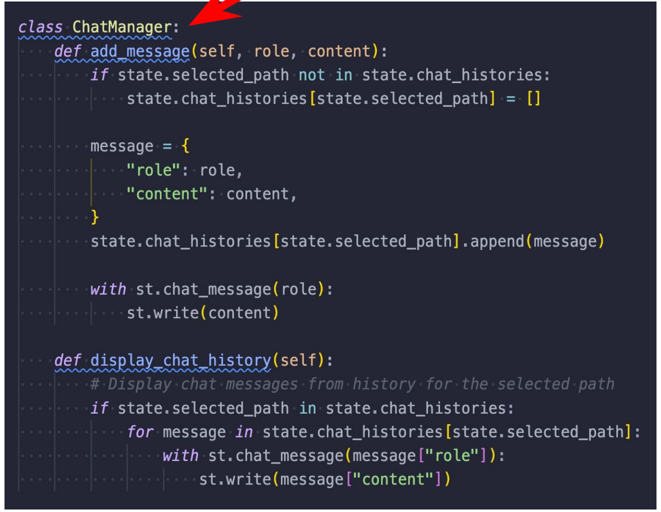
Basic Frontend Setup

Define vars



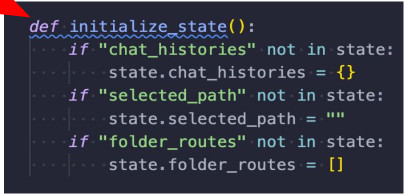
```python
import requests
import streamlit as st

API_BASE = "http://api:8000"
state = st.session_state
```

Handles saving / retrieving chat history



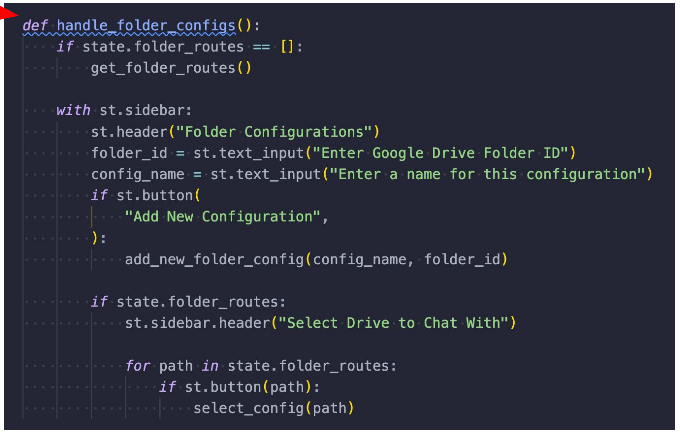
```python
class ChatManager:
    def add_message(self, role, content):
        if state.selected_path not in state.chat_histories:
            state.chat_histories[state.selected_path] = []

        message = {
            "role": role,
            "content": content,
        }
        state.chat_histories[state.selected_path].append(message)

        with st.chat_message(role):
            st.write(content)

    def display_chat_history(self):
        # Display chat messages from history for the selected path
        if state.selected_path in state.chat_histories:
            for message in state.chat_histories[state.selected_path]:
                with st.chat_message(message["role"]):
                    st.write(message["content"])
```

Initializes state



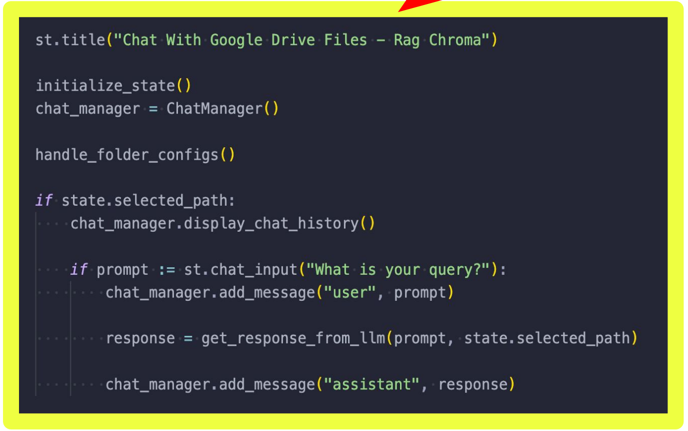
```python
def initialize_state():
    if "chat_histories" not in state:
        state.chat_histories = {}
    if "selected_path" not in state:
        state.selected_path = ""
    if "folder_routes" not in state:
        state.folder_routes = []
```

Handles selecting and displaying chains in the sidebar



```python
def handle_folder_configs():
    if state.folder_routes == []:
        get_folder_routes()

    with st.sidebar:
        st.header("Folder Configurations")
        folder_id = st.text_input("Enter Google Drive Folder ID")
        config_name = st.text_input("Enter a name for this configuration")
        if st.button(
            "Add New Configuration",
        ):
            add_new_folder_config(config_name, folder_id)

        if state.folder_routes:
            st.sidebar.header("Select Drive to Chat With")

            for path in state.folder_routes:
                if st.button(path):
                    select_config(path)
```

Our condensed app



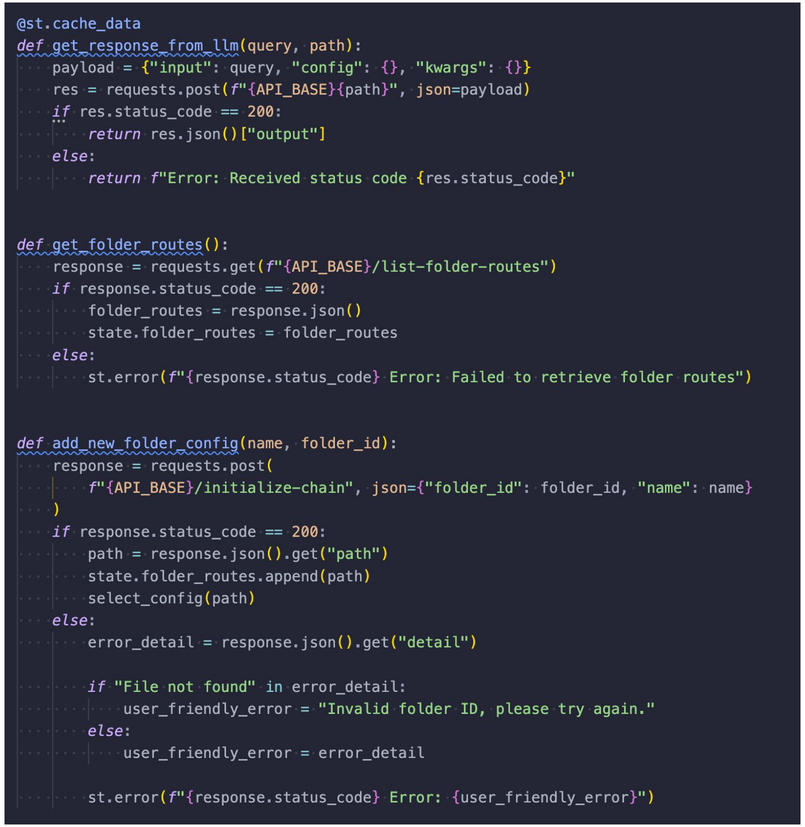
```python
st.title("Chat With Google Drive Files - Rag Chroma")

initialize_state()
chat_manager = ChatManager()

handle_folder_configs()

if state.selected_path:
    chat_manager.display_chat_history()

    if prompt := st.chat_input("What is your query?"):
        chat_manager.add_message("user", prompt)

        response = get_response_from_llm(prompt, state.selected_path)

        chat_manager.add_message("assistant", response)
```

Connecting Backend with Frontend

Query a chain

Get list of chain routes
(GET request endpoint)

Create new chain / chain endpoint
(POST request endpoint)



```python
@st.cache_data
def get_response_from_llm(query, path):
    payload = {"input": query, "config": {}, "kwargs": {}}
    res = requests.post(f"{API_BASE}{path}", json=payload)
    if res.status_code == 200:
        return res.json()["output"]
    else:
        return f"Error: Received status code {res.status_code}"


def get_folder_routes():
    response = requests.get(f"{API_BASE}/list-folder-routes")
    if response.status_code == 200:
        folder_routes = response.json()
        state.folder_routes = folder_routes
    else:
        st.error(f"{response.status_code} Error: Failed to retrieve folder routes")


def add_new_folder_config(name, folder_id):
    response = requests.post(
        f"{API_BASE}/initialize-chain", json={"folder_id": folder_id, "name": name}
    )
    if response.status_code == 200:
        path = response.json().get("path")
        state.folder_routes.append(path)
        select_config(path)
    else:
        error_detail = response.json().get("detail")

        if "File not found" in error_detail:
            user_friendly_error = "Invalid folder ID, please try again."
        else:
            user_friendly_error = error_detail

        st.error(f"{response.status_code} Error: {user_friendly_error}")
```