Chatbot documentation

# Set up

## Establish the directory of the python interpreter

In VSCode

1. **View** -> **Command Palette**
2. *Python: Select Interpreter*
3. **Enter interpreter path**
4. **Browse your file system to find a Python interpreter**
5. Browse to: *C:\Users\andrew.dilley\development\chatbot3\venv\Scripts\python.exe*

Note: can use this Python interpreter across multiple development directory structures

## Creating a new chatbot development environment

### Establish the directory structure

1. Copy the last *chatbot* directory and paste it into the *development* directory
2. Rename it to *chatbot<next number in the sequence>*
3. Delete the *venv* directory

### Create and activate the environment

1. Open a command prompt (cmd)
2. Go to the newly created development directory
3. Create the environment with python -m venv venv
4. deactivate
5. Activate the environment with *venv\Scripts\activate*

### reset the python interpreter

VSCode

View

Command palette

Python: select interpreter

Enter interpreter path

Browse

Eg C:\Users\andrew.dilley\development\chatbot11\venv\Scripts\python.exe

Select interpreter

### Verify the correct interpreter

In powershell

where python

(should return the current virtual environment)

In python shell

import sys

print(sys.executable)

### Install the dependencies

pip install -r requirements.txt

### check for any specific packages

pip show PyPDF2

### Committing a change to the code

#### Stage all modified files

In the CMD prompt and in the development directory type:

git add .

#### Commit the changes

In the CMD prompt and in the development directory type:

git commit -m "Your commit message"

data privacy on the OpenAI API Platform

**Services for businesses, such as ChatGPT Team, ChatGPT Enterprise, and our API Platform**

By default, we do not train on any inputs or outputs from our products for business users, including ChatGPT Team, ChatGPT Enterprise, and the API. We offer API customers a way to opt-in to share data with us, such as by [providing feedback in the Playground](https://help.openai.com/en/articles/9883556-providing-feedback-in-the-api-playground), which we then use to improve our models. Unless they explicitly opt-in, organizations are opted out of data-sharing by default.

Please see our [Enterprise Privacy page](https://openai.com/enterprise-privacy) for information on how we handle business data.

[How your data is used to improve model performance | OpenAI Help Center](https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance)

OpenAI encrypts all data at rest (AES-256) and in transit between our customers and us and between us and our service providers (TLS 1.2+), and uses strict access controls to limit who can access data. Our security team has an on-call rotation that has 24/7/365 coverage and is paged in case of any potential security incident. We offer a [Bug Bounty Program⁠](https://openai.com/blog/bug-bounty-program) for responsible disclosure of vulnerabilities discovered on our platform and products. Please visit our [Trust Portal⁠(opens in a new window)](https://trust.openai.com/) for more details.

[Enterprise privacy at OpenAI | OpenAI](https://openai.com/enterprise-privacy/)

[Usage policies | OpenAI](https://openai.com/policies/usage-policies/)

**Azure OpenAI Service**

But – OpenAI only has data centres in the US so we need to utilise **Azure OpenAI Service which has Australian based data centres**

**Your prompts (inputs) and completions (outputs), your embeddings, and your training data:**

* **are NOT available to other customers.**
* **are NOT available to OpenAI.**
* **are NOT used to improve OpenAI models.**
* **are NOT used to train, retrain, or improve Azure OpenAI Service foundation models.**
* **are NOT used to improve any Microsoft or 3rd party products or services without your permission or instruction.**
* **Your fine-tuned Azure OpenAI models are available exclusively for your use.**

**The Azure OpenAI Service is operated by Microsoft as an Azure service; Microsoft hosts the OpenAI models in Microsoft's Azure environment and the Service does NOT interact with any services operated by OpenAI (e.g. ChatGPT, or the OpenAI API).**

[**Data, privacy, and security for Azure OpenAI Service - Azure AI services | Microsoft Learn**](https://learn.microsoft.com/en-us/legal/cognitive-services/openai/data-privacy?utm_source=chatgpt.com&tabs=azure-portal)

* **Your organization’s data is not used to train foundation models**. Microsoft’s generative AI solutions, including Azure OpenAI Service and Copilot services and capabilities, do not use your organization’s data to train foundation models without your permission. Your data is not available to OpenAI or used to train OpenAI models.

[How-to: Create and deploy an Azure OpenAI Service resource - Azure OpenAI | Microsoft Learn](https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/create-resource?pivots=web-portal)

[Protecting the data of our commercial and public sector customers in the AI era - Microsoft On the Issues](https://blogs.microsoft.com/on-the-issues/2024/03/28/data-protection-responsible-ai-azure-copilot/?utm_source=chatgpt.com)

Docker

Build the image

cmd

docker build --tag hello-world .

view the available images

docker images

check if any docker images are running

docker ps

docker ps -a

docker start <NAME>

docker stop <NAME>

docker rm <NAME>

clear

docker logs hello

**Verify File Presence**

After rebuilding your Docker image, run the container interactively to check if the documents folder and its files exist:

bash

Copy code

docker run -it hello-world bash

Inside the container, navigate to /app/documents:

bash

Copy code

cd /app/documents

ls

Ensure Acceptable.docx and other required files are present.

**Force Rebuild Without Using Cache**

To ensure a completely fresh build (not reusing intermediate layers), you can use the --no-cache option:

bash

Copy code

docker build --no-cache --tag hello-world .

run the image

docker run -p 8080:80 –name hello -d hello-world

(8080 refers to the port onf the machine and 80 refers to the port inside of the container)

Deployment

New repository created in docker hub

cmd

docker images

docker tag hello-world andrewdilley494/hello-world

Hub

docker push andrewdilley494/hello-world

Heroku

docker build --tag chatbot8 .

docker run -p 5000:5000 --name chatbot8 -d chatbot8

docker ps -a

docker rm chatbot8

docker build -t chatbot8

git init

git add .

git commit -m "Updated chatbot using WW policy documents"

git remote add origin https://github.com/AndrewDilley/chatbot.git

git push -u origin master

OPENAI\_API\_KEY=sk-proj-aM3UrXvPa4siuqVuQCxUT5B8n\_F23Fl-9zk\_MHn6kEHlcFFKWXjmwQGWcBVkXKg4\_K0sDCd5giT3BlbkFJbsfKTNbGh9IfOldBqSSF6bAU3g7xj0iJiNCIIcrzeywe-L8UjrF0NOOrmLkpQ4OfPkOZjY0AwA

FLASK\_SECRET\_KEY=2ffc1c8235fc9e3a91d86311ef2fc1d8a370008211186f390b215d94dc7df325

FILES = ['Alcohol and Drugs in the Workplace Procedure.DOCX',

         'Consequence Of Employee Misconduct.DOCX',

         'Contractor Management Procedure.DOCX',

         'Cyber Security Incident Response Plan Framework.DOCX',

         'Flexible Working Arrangements Procedure.DOCX',

         'Gifts Benefits and Hospitality Policy - BOARD.DOCX',

         'Hazard Reporting Procedure.DOCX',

         'Incident Reporting and Response Procedure.DOCX',

         'Information Technology Security Procedure.DOCX',

         'Mobile Phone Procedure.DOCX',

         'Motor Vehicle Operational Procedure.DOCX',

         'Personal Protective Equipment and Field Uniform.DOCX',

         'Physical Security Policy.docx',

         'Use of text based Generative Artificial Intelligence (AI).DOCX',

         'Vehicle Logbook Procedure.DOCX',

         'Vehicle Safety System Alarm Procedure.DOCX',

         'Vehicle Safety System Manual.DOCX',

         'Zero Harm Policy.DOCX']

SHAREPOINT\_LINKS = {

"Alcohol and Drugs in the Workplace Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/508",

"Consequence Of Employee Misconduct.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/286",

"Contractor Management Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/417",

"Cyber Security Incident Response Plan Framework.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/885",

"Flexible Working Arrangements Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/640",

"Gifts Benefits and Hospitality Policy - BOARD.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/822",

"Hazard Reporting Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/293",

"Incident Reporting and Response Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/665",

"Information Technology Security Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/815",

"Mobile Phone Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/896",

"Motor Vehicle Operational Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/240",

"Personal Protective Equipment and Field Uniform.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/230",

"Vehicle Logbook Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/1321",

"Physical Security Policy.docx": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/1355",

"Use of text based Generative Artificial Intelligence (AI).DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/1373",

"Vehicle Safety System Alarm Procedure.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/883",

"Vehicle Safety System Manual.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/1317",

"Zero Harm Policy.DOCX": "https://wannonwater.sharepoint.com/sites/cdms/SitePages/Homepage.aspx#/PublishedDocumentView/722"

# Add other file names and links here

}

FILES = list(SHAREPOINT\_LINKS.keys())

# Update the generate\_response function

def generate\_response(user\_input):

try:

relevant\_text, file\_name = search\_relevant\_text(user\_input)

# Build the prompt

prompt = f"Use the following document text to answer the question:\n\n{relevant\_text}\n\nQuestion: {user\_input}"

# Generate a response from OpenAI

completion = client.chat.completions.create(

model="gpt-4o-mini",

messages=[

{"role": "system", "content": "You are a helpful assistant."},

{"role": "user", "content": prompt}

]

)

answer = completion.choices[0].message.content

# Add hyperlink to the reference

sharepoint\_link = SHAREPOINT\_LINKS.get(file\_name, "#")

formatted\_answer = (

f"{answer}<br><br>"

f"<span style='color:purple; font-weight:bold;'>Reference:</span> "

f"<a href='{sharepoint\_link}' target='\_blank'>{file\_name}</a>"

)

return formatted\_answer

except Exception as e:

return f"Error: {str(e)}"

**index**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Wannon Water Policy and Procedure Chatbot</title>

<link rel="stylesheet" href="{{ url\_for('static', filename='css/style.css') }}">

</head>

<body>

<h1 class="title">Wannon Water Policy and Procedure Chatbot</h1>

<div class="chat-container">

<div id="chat-box">

<!-- Chat messages will appear here -->

</div>

<form id="chat-form">

<input type="text" id="user-input" placeholder="Type your message here..." autocomplete="off" required>

<button type="submit">Send</button>

</form>

</div>

<script src="{{ url\_for('static', filename='js/script.js') }}"></script>

</body>

</html>

**Style**

.chat-container {

width: 800px; /\* Doubled width \*/

background: white;

border-radius: 8px;

box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);

overflow: hidden;

display: flex;

flex-direction: column;

}

#chat-box {

height: 600px; /\* Doubled height \*/

overflow-y: auto;

padding: 10px;

border-bottom: 1px solid #ddd;

}

.title {

text-align: center;

font-size: 2em;

color: #007bff;

margin-bottom: 20px;

}

git status

git add .

git commit -m "Updated chatbot10: linked SharePoint references"

git push origin master

chatbot11

PDF processing

Specifically the EBA

**Requirements.txt**

PyPDF2==3.0.1

**cmd**

pip install -r requirements.txt

python -m pip install setuptools

pip install --upgrade pip setuptools wheel

**dockerfile**

COPY requirements.txt /app/requirements.txt

RUN pip install -r /app/requirements.txt

**cmd**

docker build -t your-app-name .

docker run -p 5000:5000 your-app-name

pip uninstall PyPDF2

pip install PyPDF2==3.0.1

chatbot12

pre-processing the documents

should I create a new environment each time I make changes to the code?

No, you **don't need to create a new environment each time you make changes to your code**. Instead, you should:

1. **Use the Same Environment for Ongoing Development:**  
   Keep working within the same virtual environment as long as you’re making changes to the code. The environment serves as a sandbox for managing dependencies, and there’s no need to recreate it unless something major changes, such as:
   * Updating Python to a new version.
   * Major dependency changes that cause conflicts or issues.
   * The environment gets corrupted.
2. **Update the Environment When Needed:**  
   If you add new dependencies or update existing ones, install them in the current environment and update requirements.txt:
3. pip install <new-package>
4. pip freeze > requirements.txt
5. **Recreate the Environment Only When Necessary:**  
   Create a new environment if:
   * The current one becomes corrupted.
   * You switch to a new Python version.
   * You want to test a fresh setup (e.g., replicating the environment on a new machine).

**When to Create a New Environment**

* **New Project:**  
  Always create a fresh environment for new projects.
* **Major Updates:**  
  When testing a major version upgrade for Python or dependencies.
* **Corrupted Environment:**  
  If the current environment breaks, it's easier to recreate it than debug extensively.

**Good Practices**

1. **Regularly Update Dependencies:**  
   Keep your environment up-to-date with the latest versions of dependencies, ensuring compatibility and security.
2. pip list --outdated
3. pip install --upgrade <package-name>
4. **Use Version Control (Git):**  
   Keep your code changes in version control (e.g., Git). This separates code management from dependency management.
5. **Document Changes:**  
   If you add or update dependencies, make sure to document them in requirements.txt using:
6. pip freeze > requirements.txt
7. **Testing in a Fresh Environment:**  
   Occasionally test by recreating the environment from requirements.txt to ensure reproducibility:
8. python -m venv venv
9. .\venv\Scripts\activate # Activate the new environment
10. pip install -r requirements.txt

In summary, stick with your current environment for day-to-day development. Create a new environment only when testing, debugging, or starting something significantly new. Let me know if you need help managing environments! 😊

Notes:

Freezing the dependencies

The command:

bash

Copy code

pip freeze > requirements.txt

**What It Does**

This command generates a list of all the installed Python packages and their exact versions in the current environment and saves it to a file named requirements.txt.

1. **Update Regularly:**  
   Whenever you install or upgrade packages, update requirements.txt:

bash

Copy code

pip freeze > requirements.txt

1. **Avoid Unnecessary Packages:**  
   Clean up unused packages before freezing dependencies:

bash

Copy code

pip uninstall <unnecessary-package>

1. **Use a Virtual Environment:**  
   Always run this command within a virtual environment to ensure only project-specific dependencies are captured.

Check for outdated packages

The command:

pip list --outdated

**What It Does**

This command lists all the Python packages in your current environment that have newer versions available on the Python Package Index (PyPI).

**Detailed Breakdown**

1. **Installed Packages:**  
   The command examines the currently installed packages in your environment.
2. **Checks for Updates:**  
   It queries PyPI to see if newer versions are available for any installed packages.
3. **Outputs Information:**  
   The output includes:
   * The installed version of the package.
   * The latest available version.
   * The source of the installation (e.g., PyPI).

**Example Output**

Package Version Latest Type

---------- --------- --------- -----

Flask 2.1.2 2.2.2 sdist

numpy 1.24.2 1.24.3 wheel

requests 2.28.1 2.30.0 wheel

**What Each Column Means**

* **Package:** The name of the installed package.
* **Version:** The currently installed version.
* **Latest:** The newest version available on PyPI.
* **Type:** The distribution type (e.g., sdist for source distribution, wheel for binary distribution).

**Why Use It?**

1. **Keep Dependencies Up-to-Date:**
   * Identifies packages that have newer versions available to stay current with bug fixes, security patches, and new features.
2. **Resolve Compatibility Issues:**
   * If you're facing issues with a package, upgrading to the latest version might resolve them.
3. **Proactively Maintain Your Environment:**
   * Reduces technical debt by updating dependencies regularly.

**How to Update Packages**

Once you identify outdated packages, you can update them using:

**Single Package Update**

pip install --upgrade <package-name>

Example:

pip install --upgrade Flask

**Update All Outdated Packages**

To update all outdated packages at once:

pip list --outdated --format=freeze | grep -v '^\-e' | cut -d = -f 1 | xargs -n1 pip install -U

Note: Use this carefully, as it updates all packages and may introduce breaking changes.

**Best Practices**

1. **Test Updates:**  
   Before updating packages, test them in a staging environment to avoid breaking your application.
2. **Update Regularly:**  
   Run pip list --outdated periodically to ensure you’re aware of new versions.
3. **Freeze Updates:**  
   After updating, update your requirements.txt file:
4. pip freeze > requirements.txt

.reference {

    text-align: left;

    font-style: italic; /\* Italicized for distinction \*/

    margin-top: 5px;

    font-size: 0.9em; /\* Slightly smaller font \*/

}

.reference .reference-label {

    color: #333; /\* Dark gray for label \*/

    font-weight: normal;

}

.reference .reference-text {

    color: #333; /\* Forest green for data \*/

}

/\* Ensure Content enquiries label and text are styled consistently with reference \*/

.reference .content-enquiries-label {

    color: #333; /\* Match reference label color \*/

    font-weight: normal;

}

.reference .content-enquiries-text {

    color: #333; /\* Match reference text color \*/

}

Got it! To adapt the setup so the **shared volume for JSON files** is /preprocessed\_data and the **documents** are in /documents, we’ll ensure that these directories are properly mounted and used in the containers.

Here’s how the updated configuration will look:

**1. Updated preprocess\_documents.py Docker Setup**

**1.1 Updated Dockerfile**

# Dockerfile for preprocess\_documents

FROM python:3.9-slim

# Set the working directory

WORKDIR /app

# Copy the required files

COPY preprocess\_documents.py requirements.txt /app/

COPY documents /documents # Copy the documents folder to the container

# Install dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Define the shared volume

VOLUME /preprocessed\_data

# Run the script

CMD ["python", "preprocess\_documents.py", "--output-dir", "/preprocessed\_data", "--documents-dir", "/documents"]

**1.2 Command-Line Arguments in preprocess\_documents.py**

Modify preprocess\_documents.py to accept command-line arguments for directories:

import argparse

# Parse command-line arguments

parser = argparse.ArgumentParser(description="Preprocess documents to generate JSON and FAISS files.")

parser.add\_argument("--output-dir", type=str, required=True, help="Directory to save preprocessed files.")

parser.add\_argument("--documents-dir", type=str, required=True, help="Directory containing documents.")

args = parser.parse\_args()

# Use args.output\_dir and args.documents\_dir in your script logic

output\_dir = args.output\_dir

documents\_dir = args.documents\_dir

**1.3 Build and Run the Preprocessing Container**

1. Build the Docker image:
2. docker build -t preprocess-documents -f Dockerfile.preprocess .
3. Run the container:
4. docker run -v $(pwd)/preprocessed\_data:/preprocessed\_data -v $(pwd)/documents:/documents preprocess-documents

**2. Updated app.py Docker Setup**

**2.1 Updated Dockerfile**

# Dockerfile for app.py

FROM python:3.9-slim

# Set the working directory

WORKDIR /app

# Copy the required files

COPY app.py requirements.txt /app/

# Install dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Define the shared volume

VOLUME /preprocessed\_data

# Expose the application port

EXPOSE 5000

# Start the application

CMD ["python", "app.py", "--data-dir", "/preprocessed\_data"]

**2.2 Command-Line Arguments in app.py**

Modify app.py to accept a command-line argument for the data directory:

import argparse

# Parse command-line arguments

parser = argparse.ArgumentParser(description="Run the chatbot application.")

parser.add\_argument("--data-dir", type=str, required=True, help="Directory containing preprocessed files.")

args = parser.parse\_args()

# Use args.data\_dir in your app logic

data\_dir = args.data\_dir

**2.3 Build and Run the Application Container**

1. Build the Docker image:
2. docker build -t chatbot-app -f Dockerfile.app .
3. Run the container:
4. docker run -v $(pwd)/preprocessed\_data:/preprocessed\_data -p 5000:5000 chatbot-app

**3. Updated docker-compose.yml**

To orchestrate the two containers with shared volumes for /preprocessed\_data and /documents:

version: "3.8"

services:

preprocess:

build:

context: .

dockerfile: Dockerfile.preprocess

volumes:

- ./preprocessed\_data:/preprocessed\_data

- ./documents:/documents

command: ["python", "preprocess\_documents.py", "--output-dir", "/preprocessed\_data", "--documents-dir", "/documents"]

app:

build:

context: .

dockerfile: Dockerfile.app

volumes:

- ./preprocessed\_data:/preprocessed\_data

ports:

- "5000:5000"

command: ["python", "app.py", "--data-dir", "/preprocessed\_data"]

depends\_on:

- preprocess

**4. Directory Structure**

Ensure your project has the following structure:

/project

├── app.py

├── preprocess\_documents.py

├── requirements.txt

├── Dockerfile.preprocess

├── Dockerfile.app

├── docker-compose.yml

├── /documents

│ └── (Your input documents here)

└── /preprocessed\_data

└── (Generated files like text\_map.json and faiss\_index.bin)

**5. Build and Run with Docker Compose**

1. Build and run the containers:
2. docker-compose up --build
3. Check the logs to ensure:
   * preprocess\_documents.py runs and generates the files in /preprocessed\_data.
   * app.py successfully loads these files.

**6. Verify in Azure**

* **Push Docker Images**: Follow the steps outlined earlier to push images to Azure Container Registry.
* **Persistent Volume Setup**:
  + Create persistent volumes in Azure Kubernetes Service (AKS) for /preprocessed\_data and /documents.
  + Mount these volumes in both containers to ensure data sharing.

Let me know if you need assistance with the deployment to Azure or further clarification! 😊

docker run -e OPENAI\_API\_KEY="sk-proj-HdOgtUz\_C-suiU8igWdSLhhH8mpqjr4pnra3fOt7qCBBlwB9gKdeiu\_oLqZTPJTzkOW7PRVSxhT3BlbkFJu1mj9TlIDIctsI\_wC5s5hj2bMLqLSxZL1WJGtaW\_rC7omH4iVUwbUjTu7MSlHzE5kvUAhKFEIA" \

-v C:/Users/andrew.dilley/development/chatbot12/preprocessed\_data:/preprocessed\_data \

-v C:/Users/andrew.dilley/development/chatbot12/documents:/documents \

preprocess-documents

docker run -p 5000:5000 --name chatbot8 -d chatbot8

preprocess-documents

chatbot-app

docker images

docker ps -a

docker rm chatbot8

docker rm preprocess-documents

docker build --no-cache -t preprocess-documents -f Dockerfile.preprocess .

docker images

docker run -p 5000:5000 --name preprocess-documents -d preprocess-documents

docker ps -a

docker logs preprocess-documents

docker build --no-cache -t chatbot-app -f Dockerfile.app .

rm chatbot-app

run -p 5000:5000 --name chatbot-app -d chatbot-app

docker logs chatbot-app

<https://docs.docker.com/engine/storage/volumes/#use-a-volume-with-docker-compose>

**Create Volumes**

Run the following commands to create the Docker volumes:

docker volume create documents\_volume

docker volume create preprocessed\_data\_volume

**Verify Output**: Inspect the preprocessed\_data\_volume volume:

docker run --rm -v preprocessed\_data\_volume:/data busybox ls /data

**Populate the documents\_volume**

Copy your documents into the documents\_volume using the following command (update the path to your actual files):

docker run --rm -v documents\_volume:/data -v C:\Users\andrew.dilley\development\chatbot12\documents:/host\_data busybox sh -c "cp -r /host\_data/\* /data/"

**Inspect the volume contents**

docker run --rm -v documents\_volume:/data busybox ls /data

**Build the Docker Image**

Build the Docker image with the name preprocess-documents:

docker build --no-cache -t preprocess-documents -f Dockerfile.preprocess .

1 warning found (use docker --debug to expand):

- SecretsUsedInArgOrEnv: Do not use ARG or ENV instructions for sensitive data (ENV "OPENAI\_API\_KEY") (line 8)

**Check the image was created**

docker images

**run the container**

docker run --rm -v documents\_volume:/documents -v preprocessed\_data\_volume:/preprocessed\_data preprocess-documents

**Verify the Output**

After the container has run, you can inspect the preprocessed\_data\_volume to ensure the output files were generated:

docker run --rm -v preprocessed\_data\_volume:/data busybox ls /data

if os.getenv("DOCKER\_ENV") == "true":

DOCUMENTS\_PATH = "/app/documents"

PREPROCESSED\_PATH = "/app/preprocessed\_data"

else:

DOCUMENTS\_PATH = "C:/Users/andrew.dilley/development/chatbot12/documents"

PREPROCESSED\_PATH = "C:/Users/andrew.dilley/development/chatbot12/preprocessed\_data"

 **Run the Container with Interactive Shell**:

bash

Copy code

docker run --rm -it -v preprocessed\_data\_volume:/app/preprocessed\_data preprocess-documents sh

 **List Files with Timestamps**: Inside the container, navigate to the directory and use ls -l:

sh

Copy code

ls -l /app/preprocessed\_data

(venv) PS C:\Users\andrew.dilley\development\chatbot12> docker run --rm -it -v preprocessed\_data\_volume:/app/preprocessed\_data preprocess-documents sh

# ls -l /app/preprocessed\_data

total 552

-rw-r--r-- 1 root root 116781 Jan 14 06:55 faiss\_index.bin

-rw-r--r-- 1 root root 18 Jan 14 07:02 test\_file.txt

-rw-r--r-- 1 root root 441620 Jan 14 06:55 text\_map.json

**2. Debug the Script Execution**

Run the container interactively and manually check if the output directory /app/preprocessed\_data has any files:

docker run --rm -it -v documents\_volume:/app/documents -v preprocessed\_data\_volume:/app/preprocessed\_data preprocess-documents sh

at the # prompt (ie inside the container

**Check Mounted Volumes**:

* Verify the contents of the documents\_volume (mounted to /app/documents):

ls /app/documents

This should display the files in your documents\_volume.

* Yes it does!
* 

Verify the preprocessed\_data\_volume (mounted to /app/preprocessed\_data):

ls /app/preprocessed\_data

* Nothing returnedf

**Test Writing to the Preprocessed Directory**: Check if you can write files to the preprocessed\_data\_volume:

echo "test file content" > /app/preprocessed\_data/test\_file.txt

ls /app/preprocessed\_data

If this works, the volume is mounted and writable.

* This worked
* **Run Your Script Manually**: Execute your script inside the container to debug:
* sh
* Copy code
* python preprocess\_documents.py

docker build --no-cache -t app -f Dockerfile.app .

docker run --rm --env-file .env -p 5000:5000 chatbot-app

if \_\_name\_\_ == '\_\_main\_\_':

    if os.getenv("DOCKER\_ENV") == "true":

        app.run(host='0.0.0.0', port=5000, debug=False)

    else:

        app.run(host='127.0.0.1', port=5000, debug=True)

then back up to git, ignoring env

git add .gitignore

git add .

git commit -m "docker create and run working with volumes"

docker run --rm -p 80:80 app

**15 Jan**

cd development\chatbot12

*venv\Scripts\activate*

powershell

cd C:\Users\andrew.dilley\development\chatbot12

docker build --no-cache -t preprocess-documents -f Dockerfile.preprocess .

docker run --rm --env-file .env -v documents\_volume:/documents -v preprocessed\_data\_volume:/preprocessed\_data preprocess-documents

docker run --rm -v preprocessed\_data\_volume:/data busybox ls /data

**time stamps**

docker run --rm -v preprocessed\_data\_volume:/data busybox ls -lt /data

**delete files**

docker run --rm -v preprocessed\_data\_volume:/data busybox rm -rf /data/\*

**app**

docker build --no-cache -t chatbot-app -f Dockerfile.app .

docker run --rm --env-file .env -p 5000:5000 chatbot-app

next steps

add some more documents to the source

run locally first

ok ? cyber policy

**Verify Output**: Inspect the preprocessed\_data\_volume volume:

docker run --rm -v preprocessed\_data\_volume:/data busybox ls /data

**Populate the documents\_volume**

Copy your documents into the documents\_volume using the following command (update the path to your actual files):

docker run --rm -v documents\_volume:/data -v C:\Users\andrew.dilley\development\chatbot12\documents:/host\_data busybox sh -c "cp -r /host\_data/\* /data/"

**Inspect the volume contents**

docker run --rm -v documents\_volume:/data busybox ls /data

**to view the contents of the /app/preprocessed\_data folder**

docker run --rm preprocess-documents ls -l /app/preprocessed\_data

**build**

docker build --no-cache -t preprocess-documents -f Dockerfile.preprocess .

**run**

docker run --rm --env-file .env -v documents\_volume:/documents -v preprocessed\_data\_volume:/preprocessed\_data preprocess-documents

docker run --env-file .env -v documents\_volume:/documents -v preprocessed\_data\_volume:/preprocessed\_data preprocess-documents

26d74b5d2c52

docker start 26d74b5d2c52

docker exec -it 26d74b5d2c52 bash

**new directory structure**

docker run --env-file .env -v documents\_volume:/documents -v preprocessed\_data\_volume:/app/preprocessed\_data preprocess-documents

**check the volume contents**

docker run --rm -v preprocessed\_data\_volume:/data busybox ls -lt /data

this is working!

Rebuild the chatbot pointing to the right directory

**Run the chatbot**

docker run --rm --env-file .env -p 5000:5000 chatbot-app

docker run --env-file .env -v documents\_volume:/documents -v preprocessed\_data\_volume:/preprocessed\_data preprocess-documents

docker run --rm -v preprocessed\_data\_volume:/data busybox ls -lt /data

docker build --no-cache -t preprocess-documents -f Dockerfile.preprocess .

docker run --env-file .env -v documents\_volume:/documents -v preprocessed\_data\_volume:/app/preprocessed\_data preprocess-documents

docker run --rm -v documents\_volume:/data busybox ls -lt /data

docker run --rm -v preprocessed\_data\_volume:/data busybox ls -lt /data

docker run --env-file .env -v documents\_volume:/app/documents -v preprocessed\_data\_volume:/app/preprocessed\_data preprocess-documents

this works!

docker build --no-cache -t chatbot-app -f Dockerfile.app .

docker run --rm --env-file .env -p 5000:5000 -v documents\_volume:/app/documents -v preprocessed\_data\_volume:/app/preprocessed\_data chatbot-app

## Jan 16

### Migration to azure

Deploying your apps to Azure for user testing is a solid next step! Here’s a roadmap for getting these apps up and running on Azure:

**Step 1: Prepare for Azure Deployment**

1. **Ensure Environment Variables are Managed Securely**:
   * Use Azure Key Vault for securely storing sensitive information like API keys.
2. **Optimize Docker Images**:
   * Ensure your Docker images are lean and production-ready (e.g., use minimal base images like python:slim).
   * Confirm the Dockerfile exposes the correct ports (e.g., 5000 for Flask).

**Step 2: Set Up Azure Infrastructure**

1. **Choose the Deployment Service**:
   * **Azure App Service**: Simplified deployment of Docker containers with minimal setup.
   * **Azure Kubernetes Service (AKS)**: For advanced scenarios with multiple containers.
   * **Azure Container Instances (ACI)**: Lightweight container hosting without managing servers.

For testing, Azure App Service or ACI is likely sufficient.

1. **Create Azure Resources**:
   * Use the Azure Portal, Azure CLI, or Infrastructure-as-Code tools like Terraform to create:
     + **Resource Group**
     + **Container Registry** (e.g., Azure Container Registry - ACR) for hosting your Docker images.
2. **Set Up Storage for Volumes**:
   * Use Azure File Share or Azure Blob Storage to persist FAISS indexes and text maps.

**Step 3: Push Docker Images to Azure**

1. **Tag and Push Images**:
   * Log in to Azure Container Registry:
   * az acr login --name <your-acr-name>
   * Tag your Docker image:
   * docker tag <your-image> <acr-name>.azurecr.io/<image-name>:<tag>
   * Push the image:
   * docker push <acr-name>.azurecr.io/<image-name>:<tag>
2. **Verify the Image**:
   * Confirm that the image appears in your ACR.

**Step 4: Deploy the App**

1. **Using Azure App Service**:
   * Create a new Web App for Containers.
   * Configure the Web App to pull your Docker image from ACR.
   * Set environment variables in the App Service configuration (e.g., OPENAI\_API\_KEY, paths).
2. **Networking**:
   * Ensure the app is accessible via a public or private endpoint based on your needs.
3. **Attach Storage**:
   * Mount Azure File Share as a volume in the container to store FAISS indexes and text maps.

**Step 5: Test and Validate**

1. **Access the Web App**:
   * Obtain the public URL from Azure App Service.
   * Share it with testers for initial feedback.
2. **Monitor Logs and Metrics**:
   * Enable Azure Monitor for performance metrics and logging.
   * Use az webapp log tail or the Azure Portal to check logs in real time.

**Step 6: Scale and Secure**

1. **Scaling**:
   * Configure auto-scaling rules for App Service or Container Instances.
   * Set up a staging slot for testing new updates before pushing them to production.
2. **Security**:
   * Add SSL/TLS to your endpoint.
   * Restrict access to specific IPs if needed during testing.

**Next Steps**

Would you like me to:

1. Draft specific Azure CLI commands for setting up these resources?
2. Help configure your Dockerfile or container volumes for Azure?
3. Guide you on setting up Azure Key Vault or App Service?

Let me know how I can assist further! 😊

The choice between **Azure Container Registry (ACR)** and **Azure App Service** is not an "either-or" decision because they serve different purposes in the deployment process:

**Azure Container Registry (ACR)**

**Purpose:**

* **Container Image Storage**: ACR is primarily a **container registry** used to store and manage Docker container images.
* You push your built Docker images to ACR, and other Azure services (like App Service or Azure Kubernetes Service) can pull these images during deployment.

**Use Case:**

* Use ACR when:
  + You want to centrally store container images for use across multiple environments (development, staging, production).
  + You have a CI/CD pipeline that builds Docker images and pushes them to a registry for deployment.

**Pros:**

* Secure storage for container images.
* Fully integrated with Azure's security and identity (e.g., Azure Active Directory).
* Support for Helm charts (useful for Kubernetes).

**Limitations:**

* ACR doesn’t run your containers—it only stores the images.

**Azure App Service**

**Purpose:**

* **Container Hosting and Deployment**: App Service runs and manages your containerized applications. You can deploy directly from ACR or even public container registries like Docker Hub.

**Use Case:**

* Use App Service when:
  + You want a **Platform-as-a-Service (PaaS)** to easily run your containerized web apps.
  + You prefer not to manage infrastructure or a Kubernetes cluster (as would be required with AKS).
  + You need built-in scaling, monitoring, and HTTPS support.

**Pros:**

* Easy deployment from registries (ACR, Docker Hub).
* Built-in capabilities like autoscaling, logging, and SSL support.
* Minimal configuration for getting a web app live.

**Limitations:**

* Less control over infrastructure compared to Kubernetes or Virtual Machines.

**How They Work Together**

In most scenarios, you would use **ACR + App Service** together:

1. **ACR** stores your container image.
2. **App Service** pulls the image from ACR and runs the containerized application.

**When to Use Only One**

**Only ACR:**

* Rarely, if you're just using ACR as storage for your container images but deploying elsewhere (e.g., AKS or an on-prem environment).

**Only App Service:**

* If you're pulling images directly from a public registry like Docker Hub and don’t need private image storage.

**Recommendation for Your Use Case**

Since you're testing applications:

1. Use **Azure Container Registry** to securely store your Docker images.
2. Use **Azure App Service** to deploy and run your containers.

Would you like a step-by-step guide on setting up both, or help configuring App Service to pull images from ACR?

For your documents, FAISS index, and text maps, you’ll need **persistent storage in Azure** that integrates seamlessly with your applications. Here's what you can use in Azure:

**Azure Storage Options**

1. **Azure Blob Storage** (Recommended for your case)
   * **Use Case**: Storing and accessing your documents, FAISS index, and text map files.
   * **Features**:
     + Highly scalable and cost-effective object storage.
     + Supports hierarchical directory structures.
     + Access via REST APIs or SDKs.
   * **How to Use**:
     + Store raw documents (.docx, .pdf) and generated FAISS files (faiss\_index.bin and text\_map.json).
     + Configure access for your app to read/write from this storage.
2. **Azure File Share** (Alternative)
   * **Use Case**: If your application requires a mounted file system.
   * **Features**:
     + SMB/NFS protocols for mounting as a shared drive.
     + Can act as a volume for Docker containers.
   * **How to Use**:
     + Mount the file share in your Docker containers to store and access documents, index files, and maps.
3. **Azure Disk Storage** (Not ideal for your use case)
   * **Use Case**: Persistent storage for individual virtual machines or specific container workloads.
   * **Why Not**: Tied to individual resources and doesn’t offer object-based access like Blob Storage.

**Configuration for Your Application**

1. **For Document Storage**:
   * Use Azure Blob Storage to upload and store .docx and .pdf files.
   * Organize files in a container (e.g., documents).
2. **For Index and Text Map Storage**:
   * Use the same Blob Storage to store:
     + faiss\_index.bin
     + text\_map.json
   * Store them in a container (e.g., processed-data).
3. **Accessing from Applications**:
   * Use **Azure Storage SDKs** (e.g., Python's azure-storage-blob) to read and write data programmatically.
   * Configure **Shared Access Signatures (SAS)** or **Managed Identity** for secure access.
4. **Mounting as Volume (if using File Share)**:
   * Mount Azure File Share as a volume in your containerized app using the volume option in your docker-compose.yml or directly in Azure App Service.

**Steps to Set Up in Azure**

**1. Set Up Azure Blob Storage**

* Create a **Storage Account** in Azure.
* Create a **Blob Container** (e.g., documents).
* Upload your files (manual or programmatic).

**2. Set Up Access**

* Option 1: Use a **Shared Access Signature (SAS)** to give limited, time-bound access to your storage.
* Option 2: Use **Azure Managed Identity** if your app is deployed in Azure.

**3. Modify Application Code**

* Replace local file paths with Blob Storage paths.
* Use Azure SDKs (e.g., azure-storage-blob) for reading/writing.

**4. Mount as a Volume (if File Share is Used)**

* In Docker:
* volumes:
* - azure-file-share:/mnt/data
* Use Azure CLI to configure:
* az storage account create --name mystorageaccount --resource-group myresourcegroup --sku Standard\_LRS
* az storage share create --name myfileshare --account-name mystorageaccount

**Key Considerations**

* **Security**: Use Azure Key Vault to store credentials or enable Managed Identity for secure storage access.
* **Backup**: Enable backups for Blob Storage or File Share to prevent data loss.
* **Cost**: Monitor storage usage; Blob Storage has tiered pricing based on access frequency (Hot/Cold/Archive).

Would you like a detailed code example for integrating Azure Blob Storage with your app? Or guidance on setting up the storage in Azure?

Yes, **Azure Key Vault** is the final piece to ensure **secure storage and management of sensitive data** like API keys, connection strings, and other secrets. Here's how it fits into your architecture:

**Why Use Azure Key Vault?**

1. **Secure Secret Management**:
   * Store sensitive data like:
     + OpenAI API Key.
     + Azure Blob Storage connection strings or SAS tokens (if not using Managed Identity).
     + Any other configuration secrets.
2. **Centralized Access**:
   * Provides a central location for managing secrets, certificates, and encryption keys.
3. **Improved Security**:
   * Reduces hardcoding sensitive data into your application or storing it in environment variables.
4. **Integration with Azure Services**:
   * Azure services like App Service, Blob Storage, and others can securely access Key Vault secrets using **Managed Identity**.

**How to Set Up Azure Key Vault**

**1. Create a Key Vault**

* Use the Azure Portal, CLI, or PowerShell:
* az keyvault create --name <your-keyvault-name> --resource-group <your-resource-group>

**2. Add Secrets to the Key Vault**

* Store secrets such as OPENAI\_API\_KEY and STORAGE\_CONNECTION\_STRING:
* az keyvault secret set --vault-name <your-keyvault-name> --name "OPENAI\_API\_KEY" --value "<your-openai-api-key>"

**3. Grant Access to the App**

* Use a **Managed Identity** to allow your Azure App Service or Container Instance to access the Key Vault:
* az keyvault set-policy --name <your-keyvault-name> --object-id <managed-identity-object-id> --secret-permissions get
* Alternatively, use a **Service Principal** if not using Managed Identity.

**4. Access Secrets from Your Application**

* Install the Azure Key Vault SDK:
* pip install azure-keyvault-secrets azure-identity
* Update your app code to fetch secrets:
* from azure.identity import DefaultAzureCredential
* from azure.keyvault.secrets import SecretClient
* key\_vault\_name = "<your-keyvault-name>"
* kv\_url = f"https://{key\_vault\_name}.vault.azure.net/"
* credential = DefaultAzureCredential()
* client = SecretClient(vault\_url=kv\_url, credential=credential)
* openai\_api\_key = client.get\_secret("OPENAI\_API\_KEY").value

**5. Update Docker Configuration**

* If using Docker, pass the necessary environment variables to allow the container to authenticate with Azure Key Vault (e.g., enable Managed Identity or pass Azure credentials).

**Benefits of Adding Key Vault**

* **Separation of Concerns**: Secrets are not stored in code or config files.
* **Dynamic Updates**: Update secrets in Key Vault without redeploying the app.
* **Compliance**: Meets strict security and compliance requirements.

**Final Steps**

To complete your deployment:

1. Configure **Azure Key Vault** for all sensitive data.
2. Update your app to fetch secrets dynamically.
3. Test the app to ensure it correctly retrieves and uses the secrets.

Would you like a detailed walkthrough for setting up Azure Key Vault with your current apps, or help updating your code for integration?

Absolutely, Andrew! Let’s take it step by step to avoid any confusion.

**First Step: Deploy the Document and Index Storage**

The foundation of your apps is the **documents**, **FAISS index**, and **text maps**. To ensure they’re accessible to your apps in Azure, let’s set up **Azure Blob Storage**.

**Step 1: Create Azure Blob Storage**

**1. Create a Storage Account**

* Log in to the [Azure Portal](https://portal.azure.com/).
* Search for "Storage accounts" in the search bar and click **Create**.
* Fill in the required details:
  + **Resource Group**: Choose an existing one or create a new one.
  + **Storage Account Name**: Pick a unique name.
  + **Region**: Select the region closest to your users.
  + **Performance**: Choose Standard.
  + **Replication**: Start with Locally-redundant storage (LRS) for testing purposes.
* Click **Review + Create** and then **Create**.

**2. Create a Blob Container**

* Go to the newly created storage account in the portal.
* Click **Containers** under the **Data Storage** section.
* Click **+ Container**, and name it something like documents.
  + Set the **Public Access Level** to Private (recommended for security).

**3. Upload Files**

* In the documents container, click **Upload**.
* Select the .docx, .pdf, and other required files (e.g., faiss\_index.bin, text\_map.json).
* Click **Upload**.

**Step 2: Generate a Connection String for Your App**

**1. Get the Connection String**

* Go to the **Access Keys** section in your storage account.
* Copy one of the **Connection Strings**.

**2. Store the Connection String Securely**

* Temporarily, you can store it as an environment variable in your app’s Docker container.
* Later, we’ll store this in **Azure Key Vault** for better security.

**Step 3: Update Your Application to Use Azure Blob Storage**

In your app's code:

1. **Install the Azure Blob Storage SDK**:
2. pip install azure-storage-blob
3. **Add Code to Access Blob Storage**: Update your file access code to interact with Azure Blob Storage:
4. from azure.storage.blob import BlobServiceClient
5. connection\_string = "<your-connection-string>"
6. blob\_service\_client = BlobServiceClient.from\_connection\_string(connection\_string)
7. container\_name = "documents"
8. # Example: Download the FAISS index file
9. blob\_client = blob\_service\_client.get\_blob\_client(container=container\_name, blob="faiss\_index.bin")
10. with open("faiss\_index.bin", "wb") as f:
11. f.write(blob\_client.download\_blob().readall())
12. Replace local file paths in your app (like os.path) with Azure Blob Storage operations.

**Key Checkpoints for Step 1**

* **Storage Account** created and operational.
* **Blob Container** created and files uploaded.
* **Connection String** retrieved for your app.
* App code updated to fetch files from Blob Storage.

Once this step is complete, we can move on to setting up **Azure Key Vault** for secure secret management or deploying the app to Azure. Let me know how you’re progressing! 😊

Yes, Andrew, setting up a **test Azure environment** is a smart approach. Here’s why it’s beneficial and how to go about it:

**Why Use a Test Azure Environment?**

1. **Minimize Risk**:
   * Avoid potential disruptions to the production environment.
   * Prevent accidental exposure of sensitive production data or credentials.
2. **Cost Control**:
   * You can start with a minimal setup, scaling resources only when required.
   * Helps avoid unexpected charges on production subscriptions.
3. **Isolated Testing**:
   * Provides a safe space to debug and fine-tune your deployment without impacting live services.
   * Allows experimentation with configurations (e.g., security settings, permissions).
4. **Compliance**:
   * Keeps production aligned with organizational governance policies while allowing flexibility in testing.

**Options for Setting Up a Test Environment**

**1. Use a Separate Azure Subscription:**

* Ideal if your organization allows you to request a dedicated subscription for testing.
* Keeps billing and resources completely separate from production.
* You’ll need to request this from your Azure admin or IT department.

**2. Use a New Resource Group in the Existing Subscription:**

* A more straightforward option if getting a separate subscription is not feasible.
* Group all test resources (e.g., storage accounts, key vaults) under a dedicated **test resource group**.
* Easier to track costs and clean up once testing is complete.

**How to Set Up a Test Environment**

**Option 1: Separate Subscription**

1. **Request a New Subscription**:
   * If you have access to the Azure portal as an admin, create a new subscription (or request one from IT).
   * Examples: Azure Free Trial or Pay-As-You-Go subscriptions.
2. **Set Budgets and Alerts**:
   * In the test subscription, configure cost alerts to monitor spending.
3. **Duplicate Resources**:
   * Replicate necessary resources from production (e.g., storage accounts, configurations).

**Option 2: New Resource Group in Production Subscription**

1. **Create a New Resource Group**:
   * Log in to the Azure portal.
   * Click on **Resource Groups** and select **+ Add**.
   * Name it something like TestEnvironmentRG.
2. **Apply Policies**:
   * Apply tags to identify it as a test environment (e.g., Environment: Test).
   * Apply **Azure Policies** to restrict over-provisioning (e.g., VM sizes, region limits).
3. **Set a Budget**:
   * Use **Azure Cost Management** to set a budget for the test resource group.
4. **Deploy Test Resources**:
   * Create test versions of your storage account, key vault, and any other resources.

**Best Practices for the Test Environment**

* **Access Control**:
  + Limit access to the test environment to your team or testers.
  + Use least privilege principles for permissions.
* **Data**:
  + Use mock or anonymized data instead of production data.
* **Automation**:
  + If possible, use Infrastructure as Code (e.g., ARM templates, Terraform) to create and manage test resources.
* **Cleanup**:
  + Set reminders to delete or deallocate unused resources after testing.

**Next Steps**

1. Confirm whether you can request a new subscription or if you’ll use a new resource group.
2. Once set up, proceed with deploying Azure Blob Storage, Key Vault, and your apps in the test environment.

Would you like detailed instructions for setting up either option? Or help requesting a new subscription/resource group?

Here’s a comparison of using a **separate Azure subscription** versus a **new resource group in the existing subscription**, focusing on the **pros and cons**:

**Separate Azure Subscription**

**Pros:**

1. **Complete Isolation**:
   * Resources, billing, and permissions are entirely separate from production.
   * Easier to enforce stricter access controls and governance for testing.
2. **Simplified Billing**:
   * Dedicated cost tracking for testing without mixing with production expenses.
   * Useful for monitoring and allocating budgets for testing independently.
3. **Greater Flexibility**:
   * Freedom to test configurations (e.g., different subscription policies, pricing tiers) without affecting production.
4. **Compliance Support**:
   * Ideal for organizations with compliance rules requiring separation between testing and production environments.
5. **No Accidental Impact on Production**:
   * Completely eliminates the risk of accidentally modifying or misconfiguring production resources.

**Cons:**

1. **Administrative Overhead**:
   * Requires additional effort to request and set up a new subscription.
   * May need separate governance policies (e.g., cost management, security monitoring).
2. **Learning Curve**:
   * Teams may need to manage multiple subscriptions, which can add complexity to processes.
3. **Limited Integration**:
   * Integration with production services (e.g., shared resources like network or identity services) may require extra setup.
4. **Potential Cost**:
   * If it's a separate billing subscription (not tied to a shared enterprise agreement), costs might not benefit from existing production discounts.

**New Resource Group in Existing Subscription**

**Pros:**

1. **Easy Setup**:
   * Quickly create a resource group in the existing subscription without additional approval processes.
   * Familiar setup process for admins and users.
2. **Centralized Management**:
   * All resources (production and testing) are managed in the same subscription, simplifying administration.
   * Existing governance, monitoring, and security policies apply automatically.
3. **Cost-Efficiency**:
   * Testing resources can benefit from the same enterprise agreement pricing or discounts as production.
4. **Integration with Production**:
   * Easier to connect test resources to existing shared services (e.g., Azure Active Directory, virtual networks).

**Cons:**

1. **Potential Impact on Production**:
   * Mistakes or misconfigurations in test resources could inadvertently affect production (e.g., if shared networks or services are modified).
2. **Mixed Billing**:
   * Harder to separate testing costs from production unless resource tags or advanced cost tracking are used.
3. **Shared Policies**:
   * Testing resources will inherit the subscription's policies, which may limit flexibility for testing new configurations.
4. **Resource Quotas**:
   * The test environment shares the subscription’s resource quotas (e.g., cores, storage limits), which could impact production capacity.

**When to Choose Each Option**

**Separate Subscription:**

* **Recommended if**:
  + You need strict isolation for security or compliance.
  + Testing costs must be entirely separate from production billing.
  + You’re building or testing something significantly different from production.

**New Resource Group:**

* **Recommended if**:
  + You need quick and easy setup for testing.
  + There’s no strict requirement for billing separation or compliance.
  + Testing is tightly integrated with production (e.g., sharing networks, databases).

**Summary of Key Differences**

| **Feature** | **Separate Subscription** | **New Resource Group** |
| --- | --- | --- |
| **Isolation** | High | Moderate |
| **Setup Time** | Longer | Shorter |
| **Billing Separation** | Complete | Requires tags or tracking |
| **Integration** | Requires extra setup | Seamless with production |
| **Governance Flexibility** | High | Inherits subscription policies |
| **Risk to Production** | None | Minimal, but not zero |

**Recommendation**

* If **strict isolation, compliance, or clear billing separation** is a priority, go with a **separate subscription**.
* If you want **quick setup and minimal complexity**, use a **new resource group**.

Let me know which one works best for your needs, and I can guide you through the setup! 😊

Yes, your need for **access to make configuration changes** impacts the choice between a **separate Azure subscription** and a **new resource group in the existing subscription**. Let’s break this down:

**Separate Azure Subscription**

If you create a **separate subscription**, you can potentially:

1. **Gain Full Control**:
   * Be granted full **Owner** or **Contributor** rights on the subscription.
   * Allows you to manage resources, policies, and permissions without affecting production.
2. **Less Interference with Production Governance**:
   * Changes made in a separate subscription won’t conflict with the configurations, policies, or quotas applied to the production subscription.
3. **Easier Experimentation**:
   * Gives you the freedom to test without needing approval from administrators overseeing the production environment.

**Challenges:**

* You’ll need organizational approval to create a separate subscription, which might involve extra time and justification.
* If production teams rely on shared services (e.g., Azure Active Directory), setting up cross-subscription access or resources may require additional configuration.

**New Resource Group in Existing Subscription**

If you create a **new resource group** in the production subscription:

1. **Access May Be Limited**:
   * Your access will depend on the permissions granted to you for that specific resource group.
   * You’ll need to work with the Azure administrator to ensure you have adequate permissions (e.g., **Contributor** role on the resource group).
2. **Production Governance Might Apply**:
   * Any subscription-wide policies (e.g., resource locks, cost controls, or restricted regions) will also apply to the resource group, potentially limiting flexibility.
3. **Less Isolation**:
   * While resource group isolation is sufficient for many use cases, some changes (e.g., virtual network configuration or shared services) might impact other resources in the subscription.

**Advantages:**

* Easier to set up, as it doesn’t require creating a new subscription.
* Testing resources can share the same Azure Active Directory and infrastructure (e.g., networking, Key Vault) as production.

**Key Decision Points for Your Case**

| **Requirement** | **Separate Subscription** | **New Resource Group** |
| --- | --- | --- |
| **Full Configuration Control** | Easy to achieve with **Owner** role | Depends on permissions granted |
| **Minimal Impact on Production** | Fully isolated, no shared governance | May share governance and policies |
| **Ease of Setup** | Requires approval for a new subscription | Quick to set up in existing subscription |
| **Access to Production Shared Services** | Requires additional setup | Naturally integrated |
| **Flexibility for Experimentation** | High | Depends on subscription policies |

**Recommendation**

If **full control** and **freedom to experiment** are essential:

* **Go with a Separate Subscription**.
  + You can request **Owner** or **Contributor** access without impacting production governance.
  + It’s ideal if you need flexibility and isolation.

If **quick setup** and **integration with existing resources** are more important:

* **Use a New Resource Group in the Production Subscription**.
  + Work with the Azure admin to ensure you have sufficient permissions for configuration.

Would you like assistance drafting a request for admin access or help deciding which specific resources to configure once your environment is ready? 😊

Yes, if you go with a **New Resource Group in the Production Subscription**, your permissions can be **contained within that specific resource group**. This approach is often used to isolate access within a subscription, ensuring that users only have permissions for the resources they manage.

Here’s how it works:

**Permissions in a Resource Group**

1. **Role-Based Access Control (RBAC)**:
   * Azure uses **RBAC** to control permissions. Permissions can be assigned at the **resource group level**, meaning:
     + You can have full control (e.g., **Owner**, **Contributor**) over resources in the new resource group.
     + You won’t have access to resources outside that resource group unless explicitly granted.
2. **Scoped Access**:
   * Permissions assigned at the **resource group level** do not automatically extend to other resource groups or the subscription.
   * For example:
     + If you’re assigned as a **Contributor** to the test resource group, you can create, modify, and delete resources only within that group.
     + You won’t be able to view or alter resources in other groups.
3. **Inherited Permissions**:
   * Any permissions you inherit from the **subscription level** (e.g., Reader, Contributor) will also apply to all resource groups, including your test group.
   * However, if specific permissions are granted only for your resource group, they override the subscription's broader permissions for that group.

**Steps to Ensure Contained Permissions**

1. **Create the Resource Group**:
   * An administrator can create a new resource group specifically for your testing purposes (e.g., TestEnvironmentRG).
2. **Assign Scoped Permissions**:
   * The admin assigns you a role at the **resource group level**:
     + **Owner**: Full control, including assigning roles.
     + **Contributor**: Manage resources but cannot assign roles.
     + **Reader**: View-only access.

Example (Azure CLI):

az role assignment create --assignee <your-email> --role Contributor --scope /subscriptions/<subscription-id>/resourceGroups/TestEnvironmentRG

1. **Restrict Access to Other Groups**:
   * Ensure no **subscription-wide permissions** (e.g., Subscription Owner or Contributor) are assigned to you unless required.

**Advantages of Containing Permissions in a Resource Group**

* **Isolation**:
  + Your actions won’t interfere with other resource groups or production resources.
* **Granular Control**:
  + You can manage resources (e.g., storage, virtual machines) in your test group independently.
* **Accountability**:
  + Changes and resource usage are isolated, making it easy to track and manage costs and activities for the test environment.

**Considerations**

1. **Shared Resources**:
   * If your test environment needs to use shared production resources (e.g., virtual networks, Key Vault), you’ll require explicit permissions for those specific resources.
2. **Escalation Path**:
   * If you discover you need additional permissions (e.g., for creating certain resource types), you’ll need to request an update to your resource group permissions.

**Next Steps**

* Confirm with your Azure administrator that they can:
  1. Create a dedicated resource group for your testing.
  2. Assign scoped permissions (e.g., **Owner** or **Contributor**) for just that resource group.

Would you like help drafting a request to your administrator or guidance on what permissions you might need for specific tasks? 😊

Storing your **API key in Azure Key Vault** does not necessarily change the decision to use a **New Resource Group in the Production Subscription**, but it does introduce some considerations. Here's why:

**Key Vault in the Production Subscription**

1. **Key Vault Scope**:
   * Azure Key Vault is a **subscription-level resource**.
   * Even if your Key Vault resides in the **production subscription**, access to it can still be scoped to your **test resource group**.
2. **Granular Access Control**:
   * You can be granted specific access to the Key Vault (e.g., permission to retrieve the API key) without granting access to the rest of the production subscription.

Example:

* + The Azure administrator can assign you a **Key Vault Reader** or **Key Vault Secret User** role for just the test Key Vault.
  + Your app can access the secrets via **Managed Identity** without needing broader access.

1. **Shared or Separate Key Vault**:
   * **Shared Key Vault**: If production already has a Key Vault, you can store your API key there with permissions restricted to your test app.
   * **Separate Key Vault**: If desired, a new Key Vault can be created within the test resource group for your specific use.

**How Key Vault Fits with Each Option**

| **Scenario** | **Separate Subscription** | **New Resource Group** |
| --- | --- | --- |
| **Key Vault Creation** | New Key Vault required in the new subscription. | Can reuse an existing Key Vault or create a new one in the resource group. |
| **Access Control** | Full control over Key Vault in the new subscription. | Scoped access within the production Key Vault. |
| **Integration with Test App** | Needs Managed Identity or credentials from the new subscription. | Easier to integrate with shared services in the production subscription. |
| **Complexity of Setup** | Higher due to creating and configuring a new Key Vault. | Lower if reusing an existing Key Vault. |

**Does This Change the Decision?**

**No, If:**

* You are comfortable working within the **production subscription** and:
  + Key Vault access can be scoped appropriately.
  + You plan to reuse or create a Key Vault in the test resource group.

**Yes, If:**

* You want complete isolation, including Key Vault and API key storage.
* Your organization requires strict separation between production and test credentials.

**Recommendation**

For simplicity and seamless integration:

* Stick with a **New Resource Group in the Production Subscription**.
* Request a dedicated Key Vault (or access to a shared one) for your test environment.

**Next Steps**

1. **Set Up Key Vault**:
   * Decide if you’ll use an existing Key Vault or need a new one.
   * Ask your Azure admin for scoped access to the Key Vault.
2. **Integrate Key Vault with Your App**:
   * Use **Managed Identity** to securely retrieve the API key from Key Vault without hardcoding it in your app.

Would you like detailed guidance on setting up Key Vault or integrating it into your application? 😊

Yes, if your **Azure administrators are inexperienced**, getting your own **separate subscription** is a better approach for these reasons:

**Advantages of a Separate Subscription for Testing and Learning**

1. **Full Control**:
   * You’ll have complete control over resources, configurations, and permissions without depending on your administrators for setup or troubleshooting.
2. **Learning and Experimentation**:
   * You can test and refine your deployment process (e.g., Azure Key Vault, Blob Storage, Managed Identity) and document the exact steps required for a production move.
   * Experiment with different configurations without worrying about breaking shared production settings.
3. **Isolated Environment**:
   * Avoid potential conflicts with production policies, governance, or resource quotas in the production subscription.
   * Test freely without fear of impacting live resources.
4. **Documentation for Administrators**:
   * By having your own subscription, you can determine:
     + The exact resources needed.
     + The roles and permissions required.
     + How to integrate services (e.g., Key Vault, Blob Storage) effectively.
   * You can hand over precise, tested instructions to your administrators when transitioning to production.
5. **Less Dependency on Admins**:
   * You can work independently, which reduces delays caused by back-and-forth communication with inexperienced administrators.

**Challenges of a Separate Subscription**

1. **Additional Costs**:
   * A separate subscription might incur extra costs, depending on your resource usage and pricing tier.
   * Ensure that the resources you use in the test subscription align with your production budget expectations.
2. **Migration Effort**:
   * Moving to production later may require additional effort, especially if you use temporary configurations in the test subscription that aren’t compatible with production policies.
3. **Shared Services Integration**:
   * If production resources (e.g., Azure Active Directory, shared networks) are needed, integrating them across subscriptions may require coordination with your administrators.

**How This Approach Helps Prepare for Production**

1. **Refine Your Application Setup**:
   * You can validate configurations like API key storage in Azure Key Vault, permissions, and Blob Storage integration.
2. **Define Exact Requirements**:
   * Identify what resources, access, and configurations your app needs.
   * Create a step-by-step deployment guide tailored for your organization.
3. **Train Administrators**:
   * Use your tested environment as a reference to train administrators on deploying the app into production.

**Recommendation**

* If your organization allows you to request a **separate subscription**, proceed with that.
* Use the subscription to:
  + Fully test your app and deployment workflows.
  + Document the entire process, ensuring it aligns with production policies.

**Next Steps**

1. **Request a Separate Subscription**:
   * Ask for a testing subscription (e.g., Azure Pay-As-You-Go) to avoid interfering with production.
2. **Set Budget Alerts**:
   * Enable cost management tools to avoid overspending during testing.
3. **Plan for Production Migration**:
   * Design configurations (e.g., resource names, regions, access policies) that can be reused in production.

Would you like help drafting a request for a separate subscription or creating a detailed test-to-production migration plan? 😊

Requesting a **testing subscription** is not directly done through the Azure Portal in most organizations. Instead, the process typically depends on your organization's policies and Azure account management setup. Here's how you can proceed:

**Steps to Request a Testing Subscription**

**Option 1: Through Your IT/Admin Team (Most Common)**

1. **Contact Your Azure Administrator**:
   * In many organizations, Azure subscriptions are centrally managed by IT or cloud administrators.
   * Request a **new Azure subscription** for testing purposes.
2. **Provide Key Details**:
   * Purpose: Explain that the subscription is for testing and refining your application setup.
   * Scope: Highlight that it will be used for isolated, non-production workloads.
   * Budget: Specify expected resource usage (e.g., storage, compute, API calls) to address cost concerns.
3. **Approval Process**:
   * Your administrators may need to align the request with organizational policies or seek approval from management.

**Option 2: If You Have Subscription Creation Rights**

If you already have the necessary permissions, you can create a subscription directly in the Azure Portal:

1. **Sign in to the Azure Portal**:  
   [Go to Azure Portal](https://portal.azure.com/).
2. **Navigate to Subscriptions**:
   * Use the search bar to find **Subscriptions** or go to **Cost Management + Billing** > **Subscriptions**.
3. **Create a New Subscription**:
   * Click **+ Add** (or **Add subscription**).
   * Choose a subscription type (e.g., **Pay-As-You-Go**, **Microsoft Azure Dev/Test** if eligible).
4. **Provide Payment Details**:
   * If required, link it to a payment method. For enterprise agreements, your organization may handle this step.

**Option 3: Through Microsoft or Azure Marketplace**

If your organization uses the **Microsoft Customer Agreement (MCA)** or **Enterprise Agreement (EA)**:

* Contact your organization's Microsoft representative or licensing partner to request a new subscription.

**After Getting a New Subscription**

1. **Set Up a Resource Group**:
   * Organize test resources (e.g., Blob Storage, Key Vault) within the new subscription.
2. **Apply Cost Management**:
   * Set up **budgets** and **alerts** in the subscription to prevent unexpected expenses.
3. **Assign Roles**:
   * Ensure you have the **Owner** or **Contributor** role in the new subscription for full configuration control.

**Tips for Testing Subscriptions**

* **Azure Free Account**: If applicable, create a free account for testing purposes. This includes $200 in credit for the first 30 days.
* **Dev/Test Pricing**: If your organization qualifies, look into Azure Dev/Test offers, which provide discounted pricing for development and testing workloads.

Would you like help drafting a request for your IT team, or guidance on setting up your subscription once it’s available? 😊

### Change of plan – using an on-prem VM

Great, Andrew! Deploying to a local Windows-based virtual machine (VM) is a straightforward process. Here’s how to set it up step by step:

**Step 1: Prepare the Windows VM**

**1. Ensure Your VM Meets Requirements**

* Minimum specs: Sufficient CPU, RAM, and disk space to run your apps and dependencies.
* OS: Windows 10/11 Pro or a Windows Server edition (to support Docker Desktop if needed).

**2. Install Required Software**

* **Python**:
  + Download from [python.org](https://www.python.org/downloads/) and install Python 3.x.
  + During installation, check the option to add Python to the PATH.
  + Verify the installation:
  + python --version
  + pip --version
* **Docker Desktop** (optional, for containerized deployment):
  + Download and install [Docker Desktop for Windows](https://www.docker.com/products/docker-desktop).
  + Enable **WSL 2** if required during installation.
* **Git** (optional, for pulling code):
  + Download and install Git from [git-scm.com](https://git-scm.com/).

**3. Secure the VM**

* Set a strong password for the admin account.
* Configure the firewall to allow necessary ports (e.g., 5000 for Flask apps).
* Disable remote access unless explicitly needed.

**Step 2: Transfer Your Application to the VM**

**1. Copy Files**

* Use a tool like **WinSCP** or **File Explorer** (via network drive or USB) to transfer your application folder (including all code and dependencies) to the VM.

**2. (Optional) Use Git**

* Install Git on the VM.
* Clone your application repository:
* git clone https://your-repo-url.git

**Step 3: Set Up Your Application**

**1. Set Up Python Environment**

* Open Command Prompt or PowerShell on the VM.
* Create a virtual environment in the app directory:
* python -m venv venv
* Activate the virtual environment:
* venv\Scripts\activate
* Install dependencies:
* pip install -r requirements.txt

**2. Store Sensitive Information Securely**

* Add your API key to an .env file in the app directory:
* OPENAI\_API\_KEY=your\_api\_key
* Install the python-dotenv package if not already in requirements.txt:
* pip install python-dotenv

**Step 4: Run the Application**

**1. Run Without Docker**

* Start the apps using Python:
* python preprocess\_documents.py
* python app.py
* The Flask apps should start and be accessible on http://<vm-ip>:5000 and other configured ports.

**2. Run with Docker (Optional)**

* Build the Docker images locally:
* docker build -t preprocess-app:latest ./preprocess
* docker build -t main-app:latest ./app
* Start the containers:
* docker run -d -p 5000:5000 preprocess-app:latest
* docker run -d -p 5001:5001 main-app:latest

**Step 5: Enable Remote Access (If Required)**

**1. Configure Windows Firewall**

* Open **Windows Defender Firewall with Advanced Security**.
* Add inbound rules to allow traffic on the required ports (e.g., 5000 and 5001 for Flask apps).

**2. Test Local and Remote Access**

* Open a browser on the VM and navigate to http://localhost:5000.
* For remote access, navigate to http://<vm-ip>:5000 from another machine on the same network.

**Step 6: Monitor and Manage the App**

1. **Monitor Logs**:
   * If running with Python:
     + Check the console output for errors or debug messages.
   * If running with Docker:
   * docker logs <container-id>
2. **Stop or Restart**:
   * For Python apps:
     + Use Ctrl+C in the terminal to stop the app.
   * For Docker containers:
   * docker stop <container-id>
   * docker start <container-id>

**Next Steps**

Would you like detailed commands for any of these steps (e.g., Docker setup, Flask configuration)? Or help with setting up logging and monitoring? Let me know! 😊

## Securing the app

Controlling access to your app when it is running on Azure is crucial for security. Here are the key changes and configurations to manage access effectively:

**1. Use HTTPS**

* **Why**: Ensures data exchanged between users and your app is encrypted.
* **How**:
  + Azure App Service provides built-in support for HTTPS.
  + Bind a custom domain (optional) and add an SSL certificate using the Azure Portal or CLI.

**2. Configure Network Access**

**Restrict Public Access**

* Use **Access Restrictions** in Azure App Service or Azure Container Instances to allow traffic only from specific IP addresses or ranges.
  + In Azure App Service:
    1. Navigate to the App Service.
    2. Go to **Networking** > **Access Restrictions**.
    3. Add rules to permit traffic only from trusted IPs.

**Private Endpoints (Optional)**

* Use **Azure Private Link** to expose your app only to internal users or other Azure services.
  + Requires a private network setup (e.g., Azure Virtual Network).

**3. Authentication and Authorization**

**Enable Azure AD Authentication**

* Integrate Azure Active Directory (AAD) to authenticate users before accessing your app.
  + In Azure App Service:
    1. Go to **Authentication**.
    2. Enable **App Service Authentication**.
    3. Choose **Azure Active Directory** as the identity provider.
  + Requires AAD user accounts or group assignments.

**Token-Based Access**

* Use OAuth 2.0 or OpenID Connect tokens issued by Azure AD for access.
* Update your app to validate tokens in requests.

**4. API Key Management**

**Securely Store API Keys**

* Store sensitive information (e.g., OpenAI API keys) in **Azure Key Vault**.
* Use **Managed Identity** to allow the app to access the Key Vault without hardcoding secrets.

**Restrict API Key Access**

* Limit permissions of the OpenAI API key to specific actions or environments (if supported by the API provider).

**5. Limit Deployment Slot Access**

* If using deployment slots (e.g., for staging or testing), configure **slot-specific access controls**:
  + Restrict access to the staging slot by IP or require authentication.

**6. Enable Application Firewall**

* Use **Azure Front Door** or **Azure Application Gateway** with a **Web Application Firewall (WAF)**:
  + Protects your app from common threats like SQL injection, cross-site scripting (XSS), etc.
  + Can also enforce HTTPS, routing rules, and traffic monitoring.

**7. Monitor and Audit Access**

**Azure Monitor and Logs**

* Enable diagnostics and application logging:
  + Log incoming requests, authentication attempts, and errors.
  + Use Azure Monitor for real-time insights and alerts.

**Audit Access with Azure Security Center**

* Regularly review access logs and configuration audits.
* Use **Azure Security Center** to identify potential security misconfigurations.

**8. Protect Development and Test Environments**

* Ensure the same access controls are applied to test and staging environments.
* Never expose test environments to public traffic unless absolutely necessary.

**Summary of Key Changes**

| **Change** | **Tool/Service in Azure** | **Purpose** |
| --- | --- | --- |
| **Enable HTTPS** | App Service (SSL/TLS settings) | Encrypt traffic between users and the app. |
| **Restrict IP Access** | App Service > Access Restrictions | Allow only trusted IPs to access the app. |
| **Use Private Endpoints** | Azure Private Link | Limit access to internal Azure networks. |
| **Enable Authentication** | Azure Active Directory | Authenticate users before granting access. |
| **Secure API Keys** | Azure Key Vault | Store secrets securely and avoid hardcoding them. |
| **Add a WAF** | Azure Front Door/Application Gateway | Protect against common web vulnerabilities. |
| **Monitor Access** | Azure Monitor | Track and audit access activity. |

Would you like detailed guidance on setting up any of these controls or help configuring your app for Azure AD authentication? 😊

## Running the app on the VM

### Use docker

Yes, **Docker** is indeed a better option to ensure high availability for your app, especially in scenarios where reliability, consistency, and ease of management are crucial. Here's why Docker stands out and how it helps maintain high availability:

**Why Docker Ensures High Availability**

**1. Automatic Restarts**

* Docker’s --restart policies (e.g., unless-stopped or always) ensure your container:
  + Restarts automatically after a crash.
  + Comes back online after a system or Docker daemon reboot.

**2. Environment Consistency**

* Docker containers encapsulate your app and its dependencies, eliminating the risk of "it works on my machine" issues.
* This consistency reduces downtime caused by environment misconfigurations.

**3. Portability**

* You can easily move the app to another VM, cloud platform, or even a Kubernetes cluster by exporting the Docker image.

**4. Resource Isolation**

* Docker containers isolate resources like CPU, memory, and network, reducing the risk of one app interfering with another.
* This isolation prevents external factors from affecting your app's uptime.

**5. Scalability**

* Docker makes it simple to scale horizontally by running multiple containers of the app (e.g., load balancing between them).
* Works well with orchestration tools like Kubernetes or Docker Swarm.

**6. Monitoring and Logs**

* Docker provides robust logging options, making it easier to monitor the health of your app.
* Logs and container statuses can help quickly identify and resolve issues.

**Key Docker Features for High Availability**

| **Feature** | **Benefit** |
| --- | --- |
| --restart unless-stopped | Automatically restarts the app after crashes or system reboots. |
| Port Mapping (-p) | Ensures external accessibility of the app while isolating internal ports. |
| Health Checks | Define health checks in the Dockerfile to restart the app if it becomes unhealthy. |
| Volumes (-v) | Persist data outside the container, ensuring it remains safe during restarts. |
| Load Balancing | Run multiple containers with tools like Docker Compose or Docker Swarm. |

**Steps to Set Up High Availability with Docker**

**1. Prepare the App in a Dockerized Format**

* Create a Dockerfile for your app to define how the container should be built.
* FROM python:3.9
* WORKDIR /app
* COPY . /app
* RUN pip install -r requirements.txt
* CMD ["python", "app.py"]

**2. Build and Run the Container**

* Build the image:
* docker build -t app-name:latest .
* Run the container with auto-restart:
* docker run -d --restart unless-stopped -p 5000:5000 app-name:latest

**3. Test Restart Behavior**

* Stop the container manually to test the unless-stopped policy:
* docker stop <container-id>
* docker start <container-id>
* Reboot the VM and ensure the container restarts automatically:
* docker ps

**4. Add Health Checks (Optional)**

* Define a health check in your Dockerfile to monitor the app’s responsiveness:
* HEALTHCHECK --interval=30s --timeout=10s --start-period=5s CMD curl --fail http://localhost:5000 || exit 1

**5. Monitor the App**

* Use docker logs or external monitoring tools like **Prometheus** and **Grafana** to track app health.

**Future-Proofing for Production**

If you plan to scale beyond a single container, consider:

1. **Docker Compose**: Orchestrate multiple containers for your app and its dependencies (e.g., database).
2. **Load Balancing**: Use tools like NGINX or Azure Load Balancer to distribute traffic.
3. **Kubernetes**: For advanced orchestration and fault tolerance in production.

**When to Use Docker for High Availability**

* **Recommended**: If your app needs to run continuously with minimal downtime and you want a consistent, portable setup.
* **Less Suitable**: If the app is only for temporary use or does not require significant availability measures.

Would you like help with writing a Dockerfile, setting up health checks, or configuring monitoring for your Docker containers? 😊

### Use of Docker Compose

Yes, using **Docker Compose** is an excellent choice when you have multiple Dockerfiles (like Dockerfile.app and Dockerfile.preprocess) that need to work together. Docker Compose simplifies the process of managing and running multiple containers, especially when these containers depend on each other or share resources.

**Why Use Docker Compose?**

1. **Simplifies Multi-Container Management**:
   * Instead of running multiple docker run commands, you define all services in a single docker-compose.yml file.
2. **Consistent Deployment**:
   * The configuration for both containers is centralized, ensuring consistent environments across development, staging, and production.
3. **Networking Made Easy**:
   * Docker Compose automatically sets up a network for the containers, enabling seamless communication (e.g., app can call preprocess by its service name).
4. **Shared Resources**:
   * You can share volumes, networks, and environment variables between containers.
5. **Scaling**:
   * Compose makes it easy to scale individual containers (e.g., running multiple instances of app).

**Setting Up Docker Compose**

Here’s how you can set up and use Docker Compose for your Dockerfile.app and Dockerfile.preprocess.

**Step 1: Create a docker-compose.yml File**

version: '3.8' # Docker Compose version

services:

app:

build:

context: .

dockerfile: Dockerfile.app

ports:

- "5000:5000" # Map host port 5000 to container port 5000

depends\_on:

- preprocess

environment:

- APP\_ENV=development

- APP\_DEBUG=true

preprocess:

build:

context: .

dockerfile: Dockerfile.preprocess

ports:

- "5001:5001" # Map host port 5001 to container port 5001

environment:

- PREPROCESS\_ENV=production

**Step 2: Update Your Dockerfiles (if needed)**

Make sure Dockerfile.app and Dockerfile.preprocess are configured correctly:

* **Dockerfile.app**:
* FROM python:3.9
* WORKDIR /app
* COPY ./app /app
* RUN pip install -r requirements.txt
* CMD ["python", "app.py"]
* **Dockerfile.preprocess**:
* FROM python:3.9
* WORKDIR /preprocess
* COPY ./preprocess /preprocess
* RUN pip install -r requirements.txt
* CMD ["python", "preprocess\_documents.py"]

**Step 3: Build and Start Containers with Docker Compose**

1. **Build the Containers**:
2. docker-compose build
3. **Run the Containers**:
4. docker-compose up
5. **Verify Running Containers**:
   * Open your browser or use curl to test the services:
     + App: http://localhost:5000
     + Preprocess: http://localhost:5001
6. **Stop Containers**:
7. docker-compose down

**Advantages of Using Docker Compose for Your Setup**

| **Feature** | **Without Docker Compose** | **With Docker Compose** |
| --- | --- | --- |
| **Start/Stop Containers** | Multiple docker run commands | Single docker-compose up/down |
| **Networking** | Manually configure container networks | Automatic setup with service discovery |
| **Environment Management** | Configure separately for each container | Centralized in docker-compose.yml |
| **Scaling** | Complex to set up | Easy with docker-compose up --scale |

**Additional Features to Consider**

1. **Volumes**:
   * Persist data across container restarts.
   * Example in docker-compose.yml:
   * volumes:
   * - app-data:/app/data
2. **Logging**:
   * Docker Compose consolidates logs for all containers.
   * View logs with:
   * docker-compose logs
3. **Health Checks**:
   * Ensure containers are healthy before starting dependent containers.
   * Example:
   * healthcheck:
   * test: ["CMD", "curl", "-f", "http://localhost:5000"]
   * interval: 30s
   * timeout: 10s
   * retries: 3

**When to Use Docker Compose**

* **Definitely Use**:
  + When you have multiple containers (e.g., app and preprocess).
  + When containers share dependencies (e.g., databases, APIs).
  + For consistent development, testing, and deployment workflows.
* **Not Necessary**:
  + If you’re running only one container with no dependencies.

Would you like help with any specific part of this, such as configuring docker-compose.yml for your exact requirements or adding advanced features like health checks? 😊

### Getting user feedback

Adding a user rating and feedback functionality (e.g., frowning face/smiling face for ratings and optional feedback) is a great way to gather user insights. Here's how you can implement this:

**Steps to Add User Rating and Feedback**

**1. Define the Feedback Workflow**

* **Input Options**:
  + Users select either a frowning face (negative experience) or a smiling face (positive experience).
  + Optionally, they can provide written feedback.
* **Data Submission**:
  + The input is sent to your backend for processing and storage.
* **Data Storage**:
  + Store the feedback in a database (or temporary storage for testing).
* **Analysis**:
  + Aggregate ratings and feedback for analysis and improvement.

**2. Frontend Implementation**

You’ll need to update your frontend (e.g., an HTML form or modal in your app).

**HTML Example:**

html

CopyEdit

<div id="feedback-container">

<h3>Rate Your Experience</h3>

<div>

<button id="smile-button" onclick="submitFeedback('smile')">😊</button>

<button id="frown-button" onclick="submitFeedback('frown')">☹️</button>

</div>

<textarea id="feedback-text" placeholder="Optional: Provide additional feedback"></textarea>

<button onclick="submitFeedback()">Submit</button>

</div>

<script>

function submitFeedback(rating) {

const feedbackText = document.getElementById("feedback-text").value;

fetch("/feedback", {

method: "POST",

headers: {

"Content-Type": "application/json",

},

body: JSON.stringify({ rating: rating, feedback: feedbackText }),

})

.then(response => response.json())

.then(data => alert("Thank you for your feedback!"))

.catch(error => console.error("Error submitting feedback:", error));

}

</script>

**3. Backend Implementation**

Update your backend to handle feedback submissions.

**Python Flask Example:**

Add a new endpoint to accept feedback:

python

CopyEdit

from flask import Flask, request, jsonify

app = Flask(\_\_name\_\_)

feedback\_data = [] # For testing; use a database in production

@app.route('/feedback', methods=['POST'])

def feedback():

data = request.json

rating = data.get('rating') # 'smile' or 'frown'

feedback\_text = data.get('feedback', '')

# Store feedback (replace with database in production)

feedback\_entry = {"rating": rating, "feedback": feedback\_text}

feedback\_data.append(feedback\_entry)

print(f"Received feedback: {feedback\_entry}") # Debugging/logging

return jsonify({"message": "Feedback submitted successfully!"}), 200

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**4. Data Storage**

For production, use a database to store feedback. Here’s a simple schema:

**Database Table Schema:**

| **Column Name** | **Type** | **Description** |
| --- | --- | --- |
| id | INT (AUTO\_INCREMENT) | Unique identifier for each entry. |
| rating | STRING | Either smile or frown. |
| feedback | TEXT | Optional user feedback text. |
| timestamp | DATETIME | When the feedback was submitted. |

**5. Feedback Aggregation and Analysis**

Add an endpoint or script to summarize feedback:

**Example:**

python

CopyEdit

@app.route('/feedback-summary', methods=['GET'])

def feedback\_summary():

smile\_count = sum(1 for entry in feedback\_data if entry['rating'] == 'smile')

frown\_count = sum(1 for entry in feedback\_data if entry['rating'] == 'frown')

return jsonify({

"total\_feedback": len(feedback\_data),

"positive": smile\_count,

"negative": frown\_count

}), 200

**Analysis Options:**

* **Basic**: Display totals (e.g., total smiles vs. frowns).
* **Advanced**: Analyze trends over time, categorize common feedback themes, or visualize data using dashboards.

**6. Optional Enhancements**

1. **Email Notifications**:
   * Send notifications for new feedback using a service like **SendGrid** or **SMTP**.
2. **Admin Dashboard**:
   * Create a dashboard to display and analyze feedback.
3. **Authentication**:
   * Allow users to submit feedback only after logging in.
4. **Feedback Anonymity**:
   * Ensure no personal data is tied to feedback unless explicitly required.

**7. Testing**

* **Unit Tests**:
  + Test the /feedback endpoint for various inputs (e.g., valid/invalid ratings, missing feedback text).
* **User Testing**:
  + Collect feedback from a small group of testers to refine the process.

**Tools and Libraries**

* **Frontend**:
  + Use libraries like **Bootstrap** for styling the feedback form.
* **Backend**:
  + Use **SQLAlchemy** for database interaction in Python.
* **Visualization**:
  + Use **Matplotlib** or integrate with **Power BI** for advanced analysis.

Would you like a detailed guide for implementing any part, such as connecting to a database or enhancing the frontend? 😊