**Explanation and Workflow for Deployment on Azure with Docker**

Your Python application generates and saves a plot image (plt.savefig) to the filesystem during runtime, encodes the image in Base64, and then uses it in a prompt for an LLM. Here's how this can be implemented and deployed using a Docker container on Azure.

**Key Features of the Workflow**

1. **Temporary Image Storage**:
   * Images are saved temporarily during the runtime in a specified path (/content/plots).
   * These images are removed or replaced for new operations, so persistence isn't required.
2. **Base64 Encoding**:
   * The saved image is converted into a Base64 string and embedded directly into the prompt for the LLM.

**Deployment Steps**

**1. Dockerize Your Application**

# Use a Python base image

FROM python:3.12-slim

# Set the working directory inside the container

WORKDIR /app

# Copy application files to the container

COPY . /app

# Install dependencies

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

# Ensure temporary directories are created

RUN mkdir -p /content/plots

# Expose the Streamlit default port (8501)

EXPOSE 8501

# Set the command to run the Streamlit app

CMD ["streamlit", "run", "doxplor.py", "--server.port=8501", "--server.address=0.0.0.0"]

**2. Test Locally**

Build and run the Docker container locally to ensure it behaves as expected.

# Build the image

docker build -t doxplor-app .

# Run the container

docker run -it --rm -p 5000:5000 your-app

2.1 Pushing Docker image to docker hub

# Log in to Docker Hub

docker login

# Tag the image

docker tag doxplor-app <your-dockerhub-username>/doxplor-app:latest

# Push the image to Docker Hub

docker push <your-dockerhub-username>/doxplor-app:latest

**3. Deploy the Docker Image to Azure**

You can use Azure Container Instances (ACI), Azure App Service, or Azure Kubernetes Service (AKS) for deployment. Here’s an example with **Azure Container Instances (ACI)**.

1. **Push the Docker Image to Azure Container Registry**:
   * Log in to Azure:
   * az login
   * Create a container registry:
   * az acr create --resource-group your-resource-group --name your-acr-name --sku Basic
   * Log in to the registry:
   * az acr login --name your-acr-name
   * Tag and push your Docker image:
   * docker tag your-app your-acr-name.azurecr.io/your-app:latest
   * docker push your-acr-name.azurecr.io/your-app:latest
2. **Deploy to Azure Container Instances**:
3. az container create \
4. --resource-group your-resource-group \
5. --name your-container-name \
6. --image your-acr-name.azurecr.io/your-app:latest \
7. --ports 5000 \
8. --environment-variables GEMINI\_API\_KEY=your-gemini-key \
9. --cpu 1 \
10. --memory 1

**Handling Ephemeral File Storage**

In your application:

1. Save the plot in a temporary directory (/content/plots).
2. Ensure the path exists (os.makedirs(img\_path, exist\_ok=True)).
3. Remove existing images before saving new ones to avoid stale files.

**Adjustments for Cloud Deployment**

1. **Temporary File Paths**: Use /tmp/plots or /content/plots inside the container for image generation.

Update:

img\_path = '/tmp/plots' # Use a standard temporary directory in Linux-based containers

1. **Environment Variables**: Store sensitive values like API keys as environment variables in the container.

Example:

gemini\_api\_key = os.getenv("GEMINI\_API\_KEY")

1. **Application Port**: Ensure your application runs on a specified port and matches the one exposed in the container.

**Expected Behavior**

* The container runs, generates the plot, and encodes it in Base64.
* The image is sent to the LLM along with the question for analysis.
* The container's filesystem handles temporary file storage during its runtime.

**Verification in Azure**

1. Access the application endpoint provided by ACI or the Azure App Service.
2. Test the functionality by providing a schema and question to trigger the visualization and analysis workflow.

Let me know if you need assistance configuring the Docker environment or deploying on Azure!