🚀 Automating Dockerfile Generation with Ollama API

📌 Introduction

- Manually writing Dockerfiles for different programming languages can be time-consuming and error-prone. So, I built a Python script that automatically generates a best-practice Dockerfile using the Ollama API powered by the TinyLlama model. 🔥

- This project takes a programming language as input and generates a custom Dockerfile with comments, saving it for later use.

📌 Step-by-Step Guide Outline

1️⃣ Install Dependencies
- Install Python and required libraries (requests).
- Install and set up Ollama.

2️⃣ Create Project Structure
- Set up project folder with main.py, config.py, and dockerfiles/.

3️⃣ Configure API
- Define OLLAMA_API_URL and MODEL_NAME in config.py.

4️⃣ Write Python Script
- Accept programming language as input.
- Send request to TinyLlama model via Ollama API.
- Receive and save the generated Dockerfile.

5️⃣ Run the Script
- Start Ollama API (ollama serve).
- Execute python main.py.
- Enter programming language.

6️⃣ Check Output
- View generated Dockerfile in dockerfiles/ folder.

7️⃣ Troubleshooting & Enhancements
- Fix connection errors by ensuring Ollama is running.

📌 Tools Used

1️⃣ Windows Laptop
- The project was developed and tested on a Windows laptop for easy local execution.

2️⃣ ChatGPT
- Used for brainstorming ideas, debugging, and improving code efficiency.

3️⃣ Ollama (TinyLlama Model)
- A local LLM (Large Language Model) for generating Dockerfiles.

4️⃣ Python
- Core programming language used for scripting and API communication.

📌 Why Use a Local LLM?

Local LLM vs. Hosted LLM

Feature	Local LLM (TinyLlama)	Hosted LLM (API-based)
Latency	Faster, no internet dependency	May experience network delays
Privacy	Fully private, runs on own device	Data is sent to external servers
Cost	One-time setup, no API costs	Often requires **paid API access**
Customization	Can fine-tune models locally	Limited customization

Advantages of Local LLMs

✅ No Internet Required – Works offline.
✅ No API Costs – No need to pay for API usage.
✅ Better Data Privacy – Keeps all processing on the local machine.

Disadvantages of Local LLMs

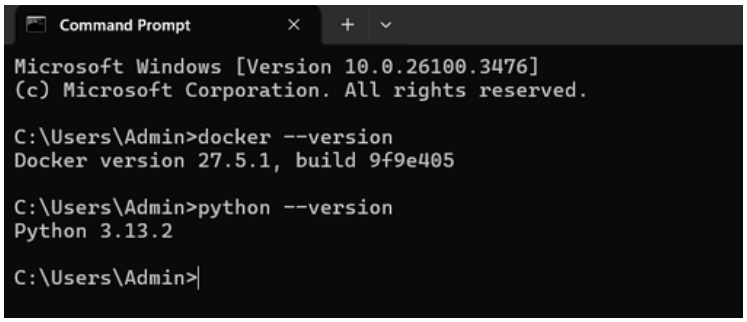
❌ Hardware Dependent – Requires a good CPU/GPU.
❌ Limited Model Size – Cannot use very large models.
❌ Manual Setup – Needs installation and configuration.

This project leverages a local LLM to ensure fast, cost-effective, and private Dockerfile generation without relying on external APIs. 🚀

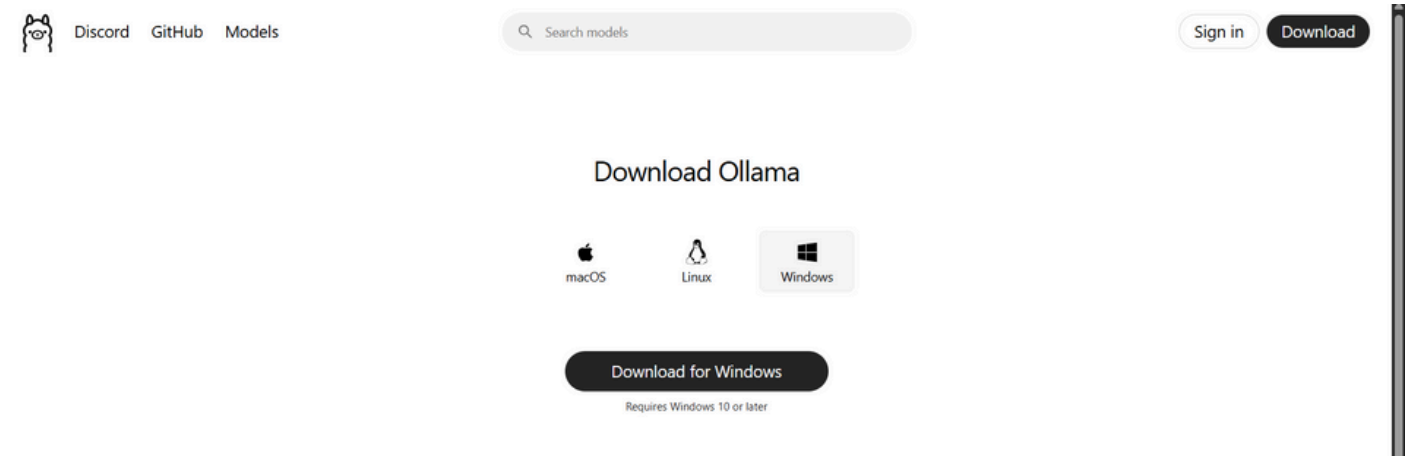
Step 1: Install Required Software

Open CMD and check Python and Docker are installed correctly and working or not
By typing

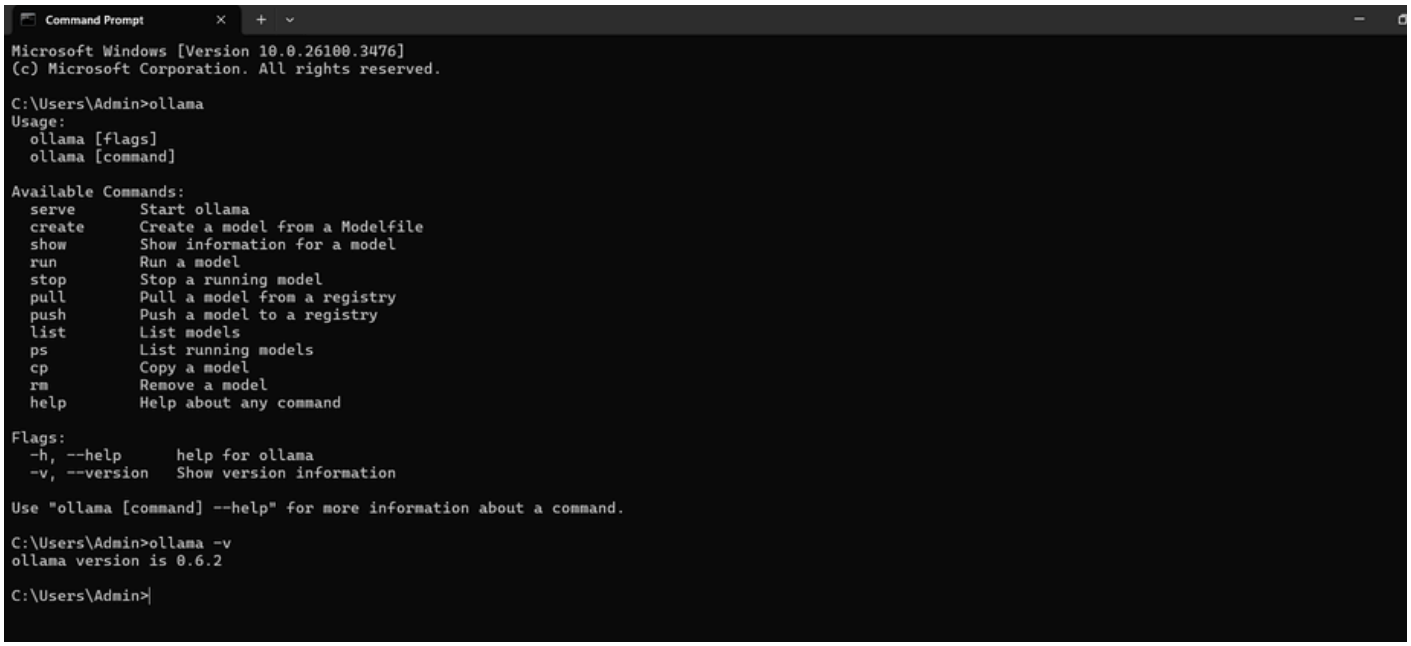
1. python --version
2. docker --version



Download Ollama windows version form the its website and install it



Check if its correctly instaled or not by typing “ollama” on cmd. and check version by “ ollama --version”



Step 2: Setup Your Project Folder

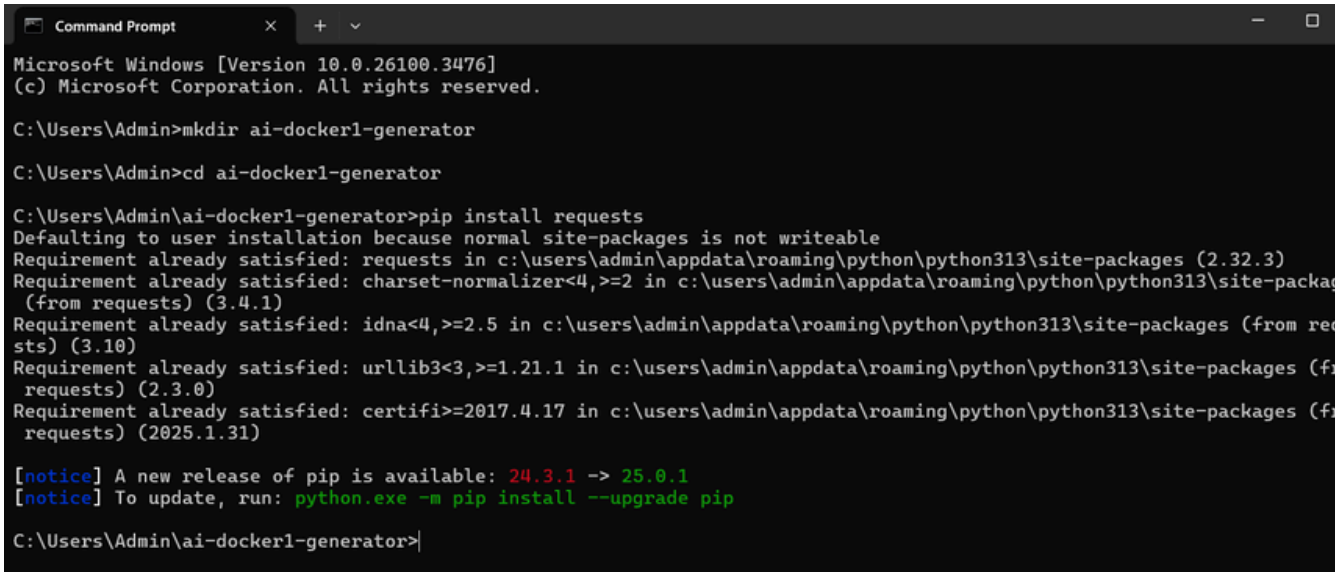
Create a new folder for your project by typing the below commands

“mkdir ai-docker-generator”
“cd ai-docker1-generator”

Step 3: Install Python Dependencies

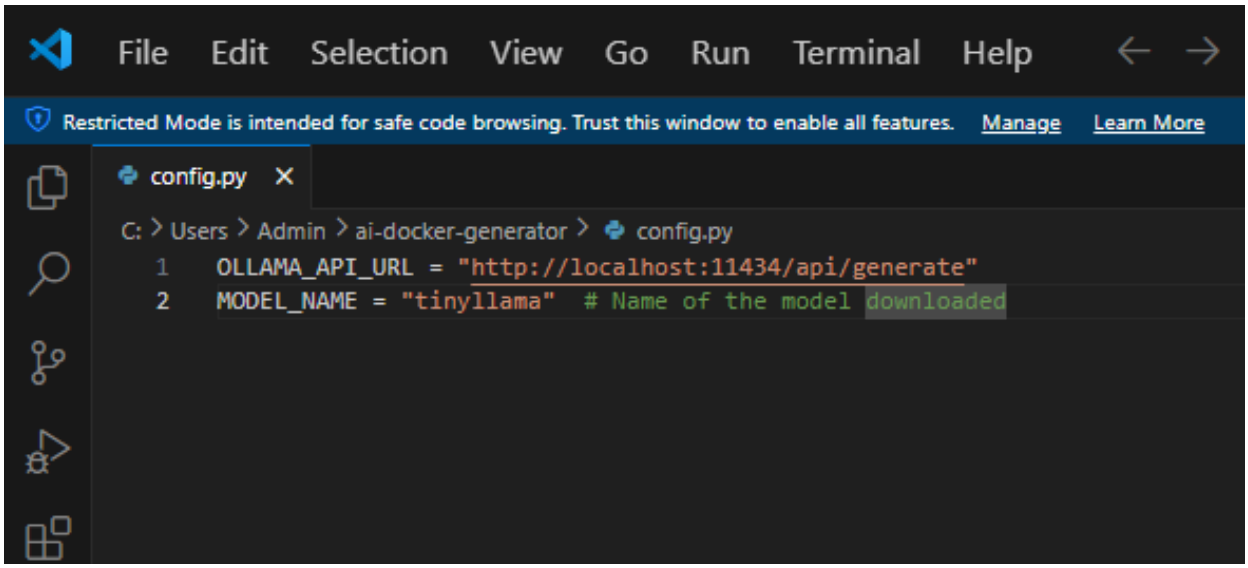
Inside the project folder, install required libraries:

“pip install requests”



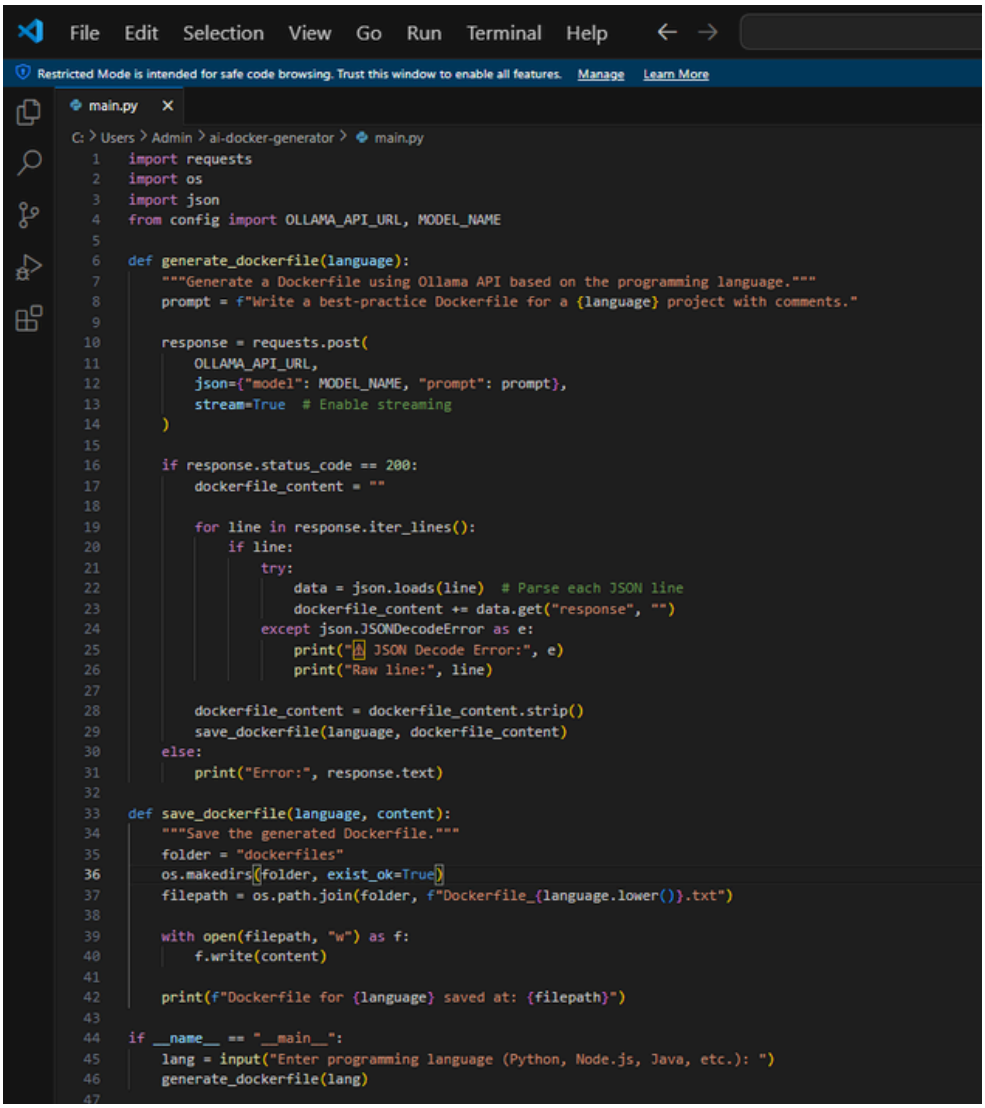
Step 4: Create Configuration File

Inside your project folder, create a file named config.py and add the Ollama API URL:

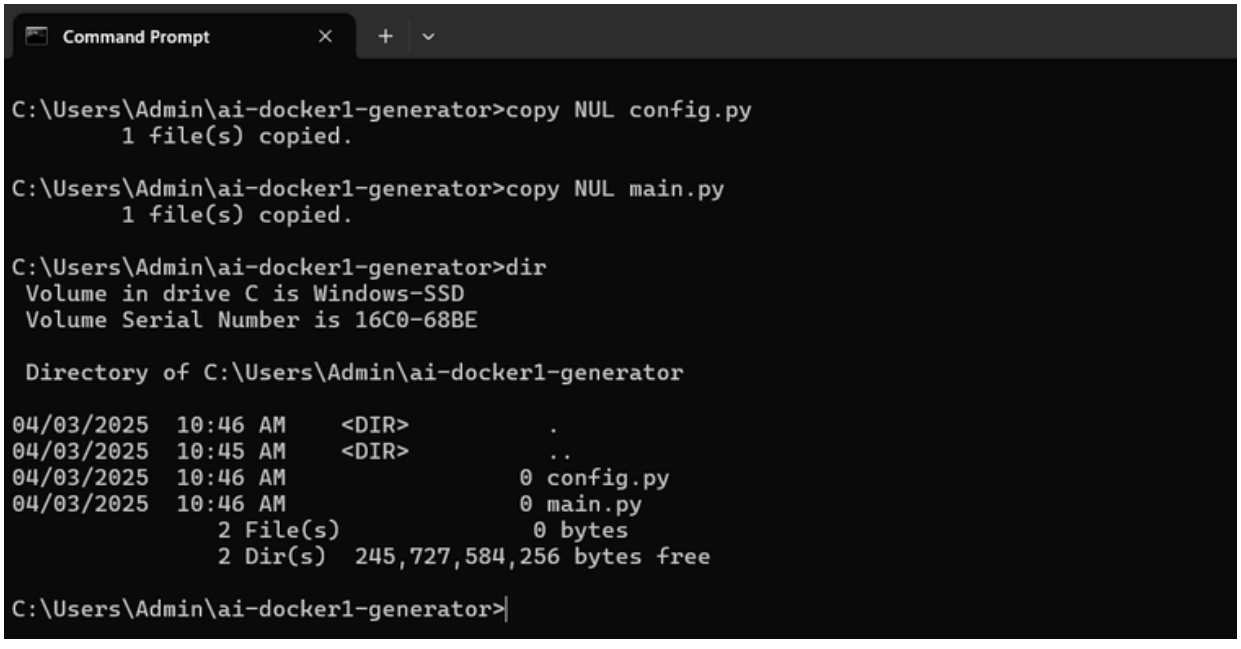


Step 5: Create the Main Script

Create a file main.py inside the folder.



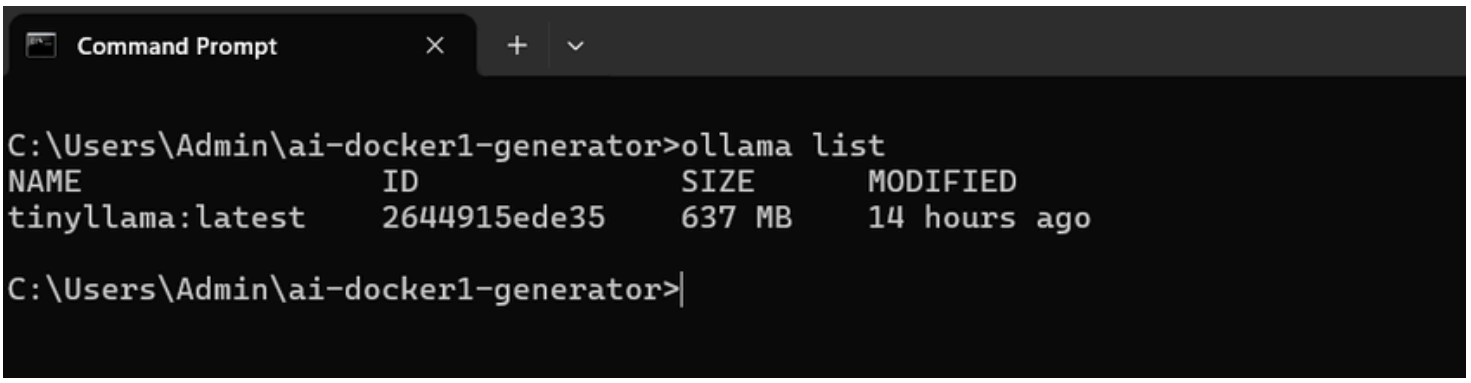
Here we can see both the file are present in the folder ai-docker1-generator



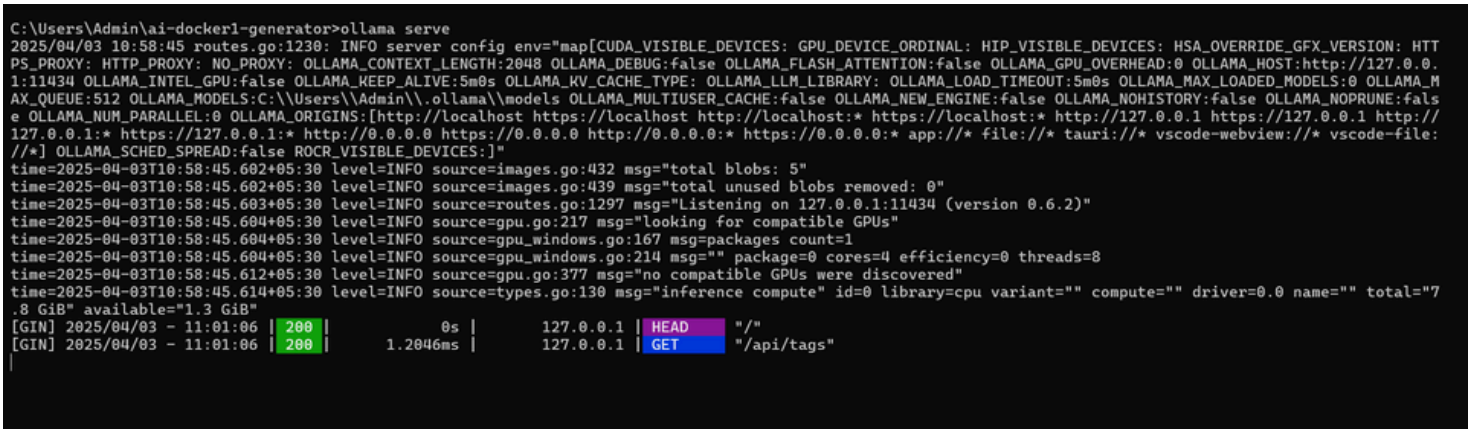
Step 6: Pull a Ollama Model and start Ollama API server

For my project I have downloaded Ollama Model - “Tinyllama” Since it requires less ram for the process and then start the ollama api server

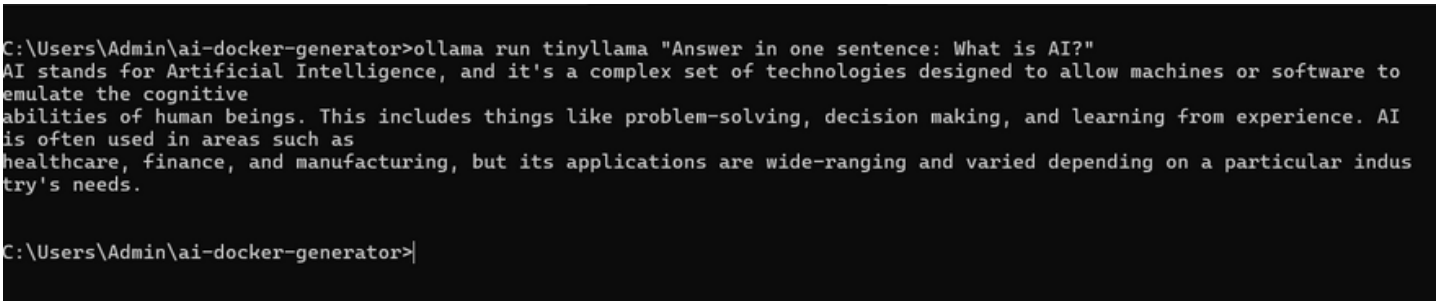
“ollama pull tinyllama”
“ollama serve”
since i have alredy downloaded i can check my model by typing “ollama list”



After “ollama serve” command we can see the api server begins



check weather the model is working and reponding properly or not by simply asking a question
“ollama run tinyllama "Answer in one sentence: What is AI?"

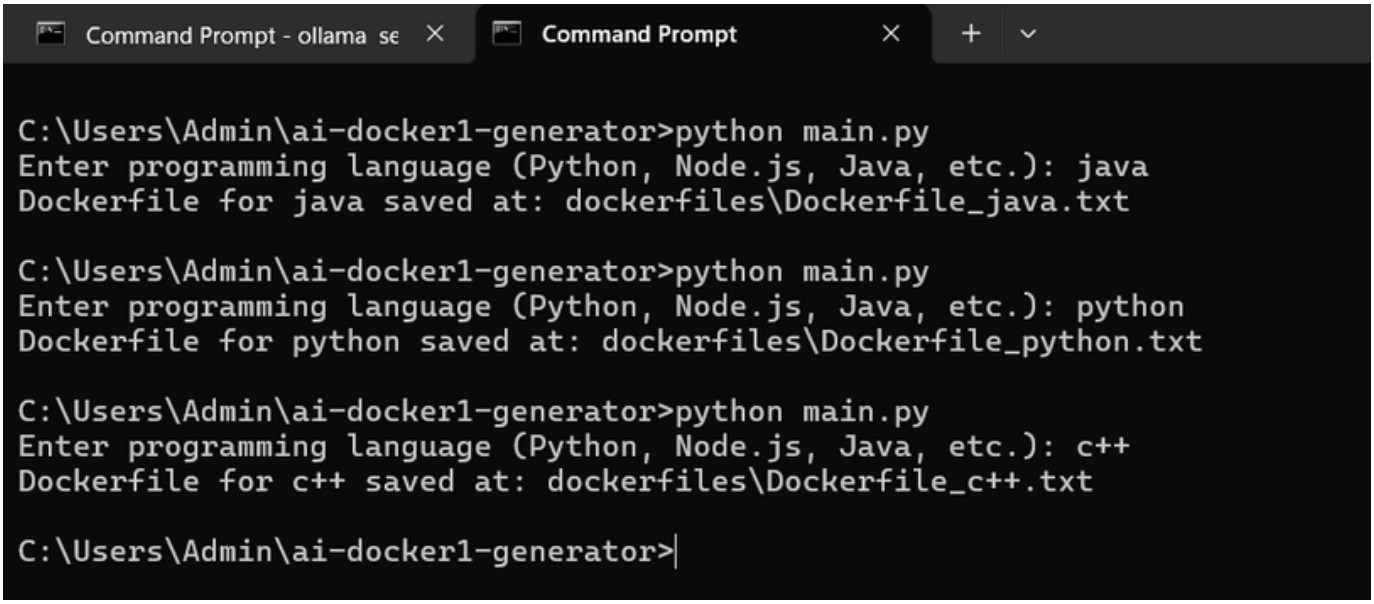


Respond correctly. 👍

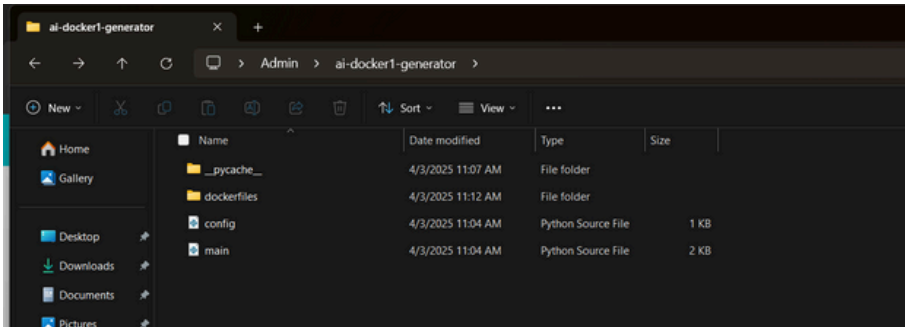
Step 7: Running the Script main.py

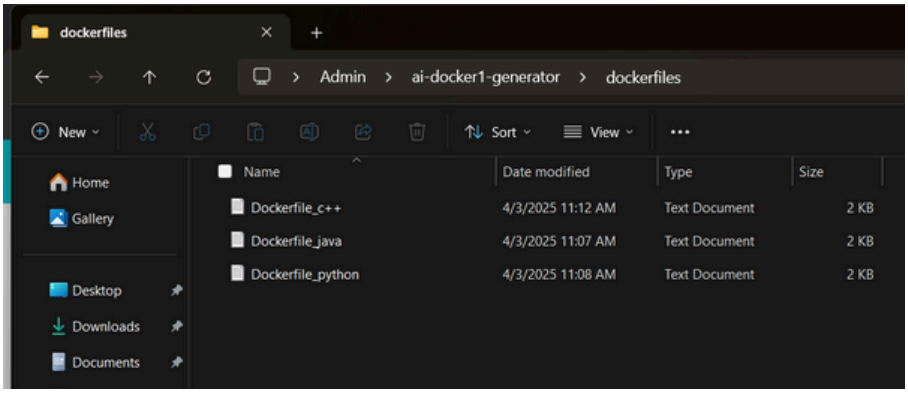
Run the script from the terminal:

“python main.py”



After running the main file we get a question to choose a language to generate the docker file.
for reference I have done for Python, Java, C++ Languages

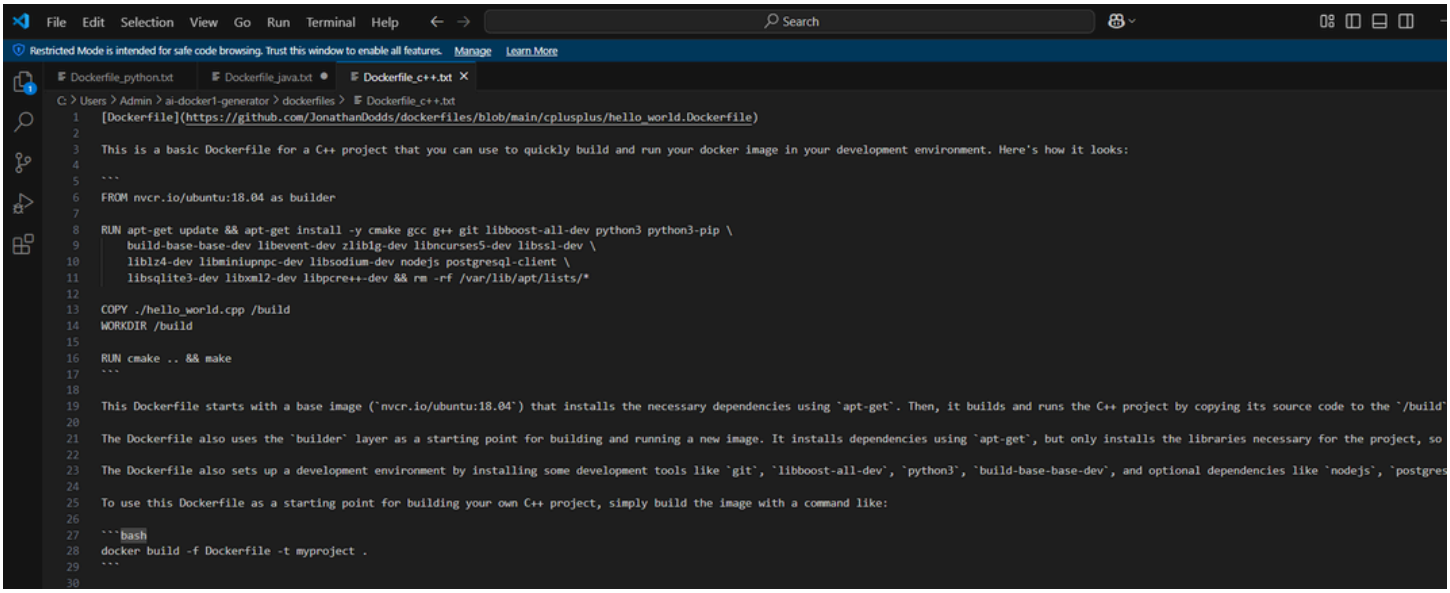




We can see that the model had automatically created a Docker folder and started adding files after creating them

Below are the Docker files created By Ollama Tinyllama Model

For C++



```
[Dockerfile](https://github.com/JonathanDodds/dockerfiles/blob/main/cplusplus/hello_world.Dockerfile)

This is a basic Dockerfile for a C++ project that you can use to quickly build and run your docker image in your development environment. Here's how it looks:

```
FROM nvcr.io/ubuntu:18.04 as builder

RUN apt-get update && apt-get install -y cmake gcc g++ git libboost-all-dev python3 python3-pip \
    build-base-base-dev libevent-dev zlib1g-dev libncurses5-dev libssl-dev \
    liblz4-dev libminiupnpc-dev libsodium-dev nodejs postgresql-client \
    libsqlite3-dev libxml2-dev libpcre++-dev && rm -rf /var/lib/apt/lists/*

COPY ./hello_world.cpp /build
WORKDIR /build

RUN cmake .. && make
```

This Dockerfile starts with a base image (`nvcr.io/ubuntu:18.04`) that installs the necessary dependencies using `apt-get`. Then, it builds and runs the C++ project by copying its source code to the `/build`

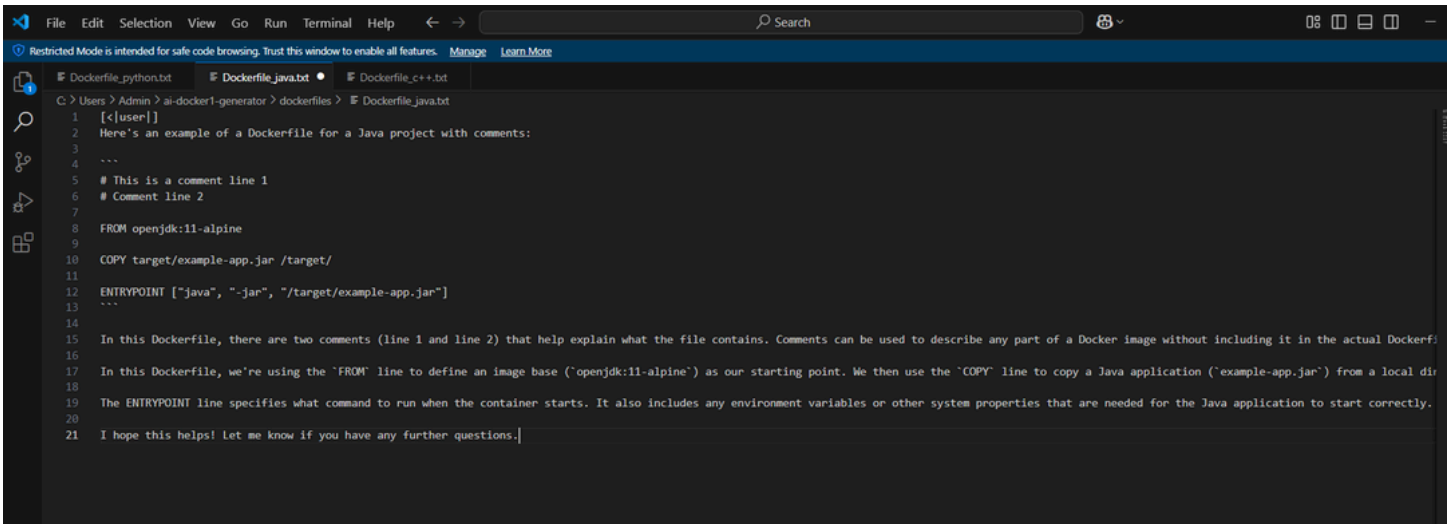
The Dockerfile also uses the `builder` layer as a starting point for building and running a new image. It installs dependencies using `apt-get`, but only installs the libraries necessary for the project, so

The Dockerfile also sets up a development environment by installing some development tools like `git`, `libboost-all-dev`, `python3`, `build-base-base-dev`, and optional dependencies like `nodejs`, `postgres

To use this Dockerfile as a starting point for building your own C++ project, simply build the image with a command like:

```bash
docker build -f Dockerfile -t myproject .
```
```

For JAVA



```
[<|user|]
Here's an example of a Dockerfile for a Java project with comments:

```
# This is a comment line 1
# Comment line 2

FROM openjdk:11-alpine

COPY target/example-app.jar /target/

ENTRYPOINT ["java", "-jar", "/target/example-app.jar"]
```

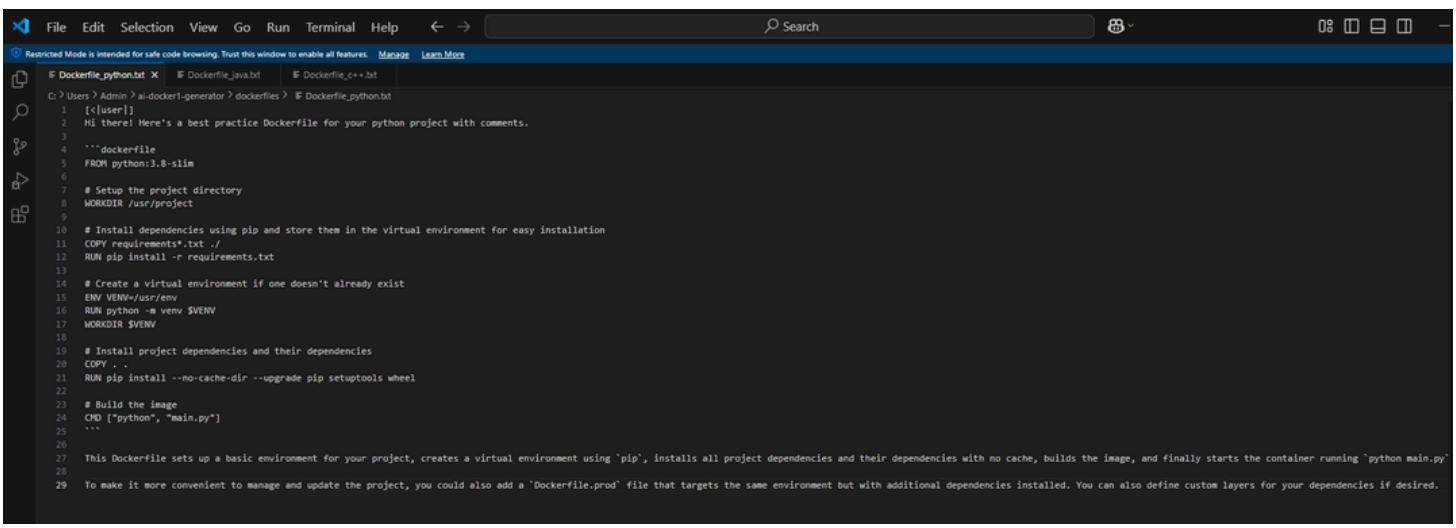
In this Dockerfile, there are two comments (line 1 and line 2) that help explain what the file contains. Comments can be used to describe any part of a Docker image without including it in the actual Dockerfi

In this Dockerfile, we're using the `FROM` line to define an image base (`openjdk:11-alpine`) as our starting point. We then use the `COPY` line to copy a Java application (`example-app.jar`) from a local dir

The ENTRYPOINT line specifies what command to run when the container starts. It also includes any environment variables or other system properties that are needed for the Java application to start correctly.

I hope this helps! Let me know if you have any further questions.
```

For Python



```
[<|user|]
Hi there! Here's a best practice Dockerfile for your python project with comments.

```dockerfile
FROM python:3.8-slim

# Setup the project directory
WORKDIR /usr/project

# Install dependencies using pip and store them in the virtual environment for easy installation
COPY requirements*.txt ./
RUN pip install -r requirements.txt

# Create a virtual environment if one doesn't already exist
ENV VENV=/usr/env
RUN python -m venv $VENV
WORKDIR $VENV

# Install project dependencies and their dependencies
COPY . .
RUN pip install --no-cache-dir --upgrade pip setuptools wheel

# Build the image
CMD ["python", "main.py"]
```

This Dockerfile sets up a basic environment for your project, creates a virtual environment using `pip`, installs all project dependencies and their dependencies with no cache, builds the image, and finally starts the container running `python main.py`

To make it more convenient to manage and update the project, you could also add a `Dockerfile.prod` file that targets the same environment but with additional dependencies installed. You can also define custom layers for your dependencies if desired.
```

Conclusion

DevOps is all about automation, and integrating AI into the process makes automation even more efficient, reliable, and faster. By using Ollama's TinyLlama model, I was able to generate Dockerfiles dynamically, reducing manual effort and improving consistency in containerization.

While the generated Dockerfile may not be perfect, it can be significantly improved with better model training and fine-tuning. The potential of AI in DevOps automation is vast, and with continuous improvements, it can become an essential tool for streamlining infrastructure management.

For scripting, I leveraged ChatGPT to assist in writing the Python script that interacts with the Ollama API. While the project itself is simple, setting up Ollama, troubleshooting installation issues, and fine-tuning model responses required patience and problem-solving skills.

This project was a great learning experience, demonstrating how AI-powered automation can optimize DevOps workflows while also highlighting the challenges of working with local AI models. 🚀