



LLM Application Development

Setup Guide & Code Walkthrough

Virtual Environment • Poetry • LangChain • Google Gemini • Streamlit

1. Virtual Environment Setup

1.1 What is a Virtual Environment?

A virtual environment is an isolated Python installation created specifically for one project. It has its own copy of Python and its own set of installed packages — completely separate from any other Python project or your system Python installation.

Without a virtual environment, all your Python projects share the same global package pool. This causes version conflicts — Project A needs pandas 1.5 while Project B needs pandas 2.2. Virtual environments solve this completely by giving each project its own private space.

- **Isolation:** packages installed for this project won't affect any other project.
- **.venv folder:** the virtual environment lives in a hidden .venv folder inside your project directory.
- **Activation:** you must activate the virtual environment before installing or running anything — your terminal prompt will show (.venv) when active.

💡 Key Point: Always create a virtual environment before starting any new Python project. It is the single best practice that prevents 99% of 'it works on my machine' issues.

1.2 Prerequisites

Before creating the environment, ensure these tools are installed on your machine:

- **Python 3.12:** download from <https://www.python.org/downloads/> — tick 'Add Python to PATH' during installation on Windows.
- **VSCode:** download from <https://code.visualstudio.com/> — the recommended code editor for this guide.
- **Google API Key:** required to call Google Gemini. Obtain from <https://aistudio.google.com/app/apikey>.

1.3 Create the Virtual Environment

Open your project folder in a terminal (VSCode Terminal: View → Terminal) and run the commands for your operating system:

Windows — VSCode PowerShell Terminal

```
# Create a virtual environment named .venv using Python 3.12
py -3.12 -m venv .venv

# Activate the virtual environment
.\.venv\Scripts\activate


# If you get a security/execution policy error, run this first:
Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process

# Then try activating again:
.\.venv\Scripts\activate
```

Mac / GCP Cloud Shell — Bash Terminal

```
# Create a virtual environment named .venv using Python 3.12
python3.12 -m venv .venv

# Activate the virtual environment
source .venv/bin/activate
```


 **Note:** After activation, your terminal prompt changes to show (.venv) at the beginning — e.g. (.venv) PS C:\my_project>. This confirms the virtual environment is active. Every pip or poetry command now applies only to this project.

1.4 Create the .env File

Store API keys and project credentials in a .env file inside your project root. Never hardcode sensitive values directly in Python files — the python-dotenv library reads this file and injects the values as environment variables at runtime.

Create a file named .env in your project root directory:

```
# .env — fill in your actual values
GOOGLE_API_KEY=your_google_api_key_here
GCP_PROJECT_ID=your_project_id
GCP_REGION=us-central1
```

 **Note:** Add .env to your .gitignore file so API keys are never accidentally committed to version control. Create a .gitignore file and add the line: .env

2. Poetry — Dependency Management


2.1 What is Poetry?

Poetry is a modern Python tool for dependency management and packaging. It replaces the old combination of pip + requirements.txt + setup.py with a single, clean workflow. Poetry automatically creates isolated virtual environments, tracks exact package versions in a lock file, and ensures your project installs and runs identically on every machine and environment.

- **pyproject.toml**: the single configuration file for your entire project. Contains the project name, Python version constraints, and all package dependencies. Replaces requirements.txt, setup.py, and setup.cfg.
- **poetry.lock**: automatically generated — records the exact versions of every installed package and all of their transitive dependencies. Like package-lock.json in Node.js. Guarantees reproducible builds across machines.

Why Poetry over pip + requirements.txt?

Feature	pip + requirements.txt	Poetry
Dependency resolution	Basic — can conflict	Advanced — resolves all conflicts automatically
Virtual environment	Manual (python -m venv)	Automatic — created and managed by Poetry
Exact version locking	Optional (pip freeze > requirements)	Built-in — poetry.lock always generated
Add / remove packages	pip install / uninstall manually	poetry add / remove — updates everything atomically
Project config file	requirements.txt (flat package list)	pyproject.toml (full project metadata)
Run scripts in env	Activate venv, then python script.py	poetry run python script.py (no activation needed)
Reproducible builds	Not guaranteed	Guaranteed via poetry.lock

 **Key Point:** Always commit both pyproject.toml AND poetry.lock to version control (Git). This ensures every team member gets exactly the same package versions when they run 'poetry install'.

2.2 Project Folder Structure

After initialising Poetry and creating your files, your project directory should look like this:

```
my_project/
|
|-- pyproject.toml      # Project metadata + dependencies
|-- poetry.lock         # Exact pinned versions of all packages
|-- .env               # API keys (never commit to Git!)
|-- .gitignore         # Should include: .env and .venv/
|-- .venv/             # Isolated virtual environment folder
|-- 1_llm_call.py       # Script 1: basic LLM call
|-- 2_llm_personalized_chatbot.py # Script 2: terminal chatbot
|-- 3_llm_with_ui.py    # Script 3: Streamlit web UI
```

2.3 Install Poetry

With your virtual environment activated (you should see (.venv) in your terminal prompt), install Poetry using pip:

```
# Install Poetry inside the activated .venv
pip install poetry

# Verify the installation was successful
poetry --version
▶ Output: Poetry (version 2.x.x)
```

2.4 Initialise Poetry in Your Project

Navigate to your project folder and initialise Poetry. This creates the pyproject.toml file which will track all project dependencies going forward.

For a brand new project — creates the folder and pyproject.toml

```
poetry new my_project
cd my_project
```

For an existing project folder — adds pyproject.toml to current directory

```
poetry init
```

💡 **Key Point:** Use 'poetry init' when you already have Python files and just want to add Poetry management. Use 'poetry new' to start a completely fresh project from scratch.

2.5 Install Project Dependencies

Add all required packages using 'poetry add'. Each command installs the package, adds it to pyproject.toml, and regenerates poetry.lock with the exact pinned version.

Install each library individually:

```
# LangChain — core framework for building LLM applications
poetry add langchain==0.3.27

# python-dotenv — loads .env file variables into os.environ
poetry add python-dotenv==1.1.1

# LangChain integration for Google Gemini API
poetry add langchain-google-genai==2.1.12

# NumPy — numerical computing (required by pandas and ML libs)
poetry add numpy==2.2.0

# Pandas — data manipulation and analysis
poetry add pandas==2.2.2

# Streamlit — build interactive web UIs in pure Python
poetry add streamlit==1.50.0
```

Or install all packages in a single command:

```
poetry add langchain==0.3.27 python-dotenv==1.1.1 langchain-google-genai==2.1.12
numpy==2.2.0 pandas==2.2.2 streamlit==1.50.0
```

Verify all packages are installed:

```
poetry show
```

2.6 Library Reference

Library	Version	Purpose	Why We Need It
langchain	0.3.27	Core LangChain framework	Standard interface to interact with any LLM provider
python-dotenv	1.1.1	Loads .env variables into os.environ	Safely reads API keys — no hardcoding in source code
langchain-google-genai	2.1.12	LangChain integration for Google Gemini	Provides the ChatGoogleGenerativeAI class for Gemini calls
numpy	2.2.0	High-performance numerical array operations	Core dependency required by pandas and ML libraries
pandas	2.2.2	DataFrame-based data manipulation and analysis	For data loading and processing in data science projects
streamlit	1.50.0	Interactive web apps in pure Python	Wraps the LLM call in a browser-based UI — no HTML needed

2.7 Poetry Command Reference

Command	Description
pip install poetry	Install Poetry (run once inside activated .venv)
poetry new my_project	Create a new project with standard Poetry structure
poetry init	Add Poetry to an existing project directory
poetry add pandas	Install a package and add it to pyproject.toml
poetry add pandas==2.2.2	Install a specific pinned version of a package
poetry remove pandas	Uninstall a package and remove it from pyproject.toml
poetry install	Install all packages from poetry.lock (for reproducing exact environment)
poetry show	List all currently installed packages and versions
poetry shell	Open a new shell session with the virtual environment activated
poetry run python script.py	Run a Python file inside the virtual environment (no activation needed)
poetry run streamlit run app.py	Run a Streamlit app inside the virtual environment
poetry update	Update all packages to their latest compatible versions
poetry publish --build	Build and publish the project package to PyPI

3. What is an LLM?

3.1 Large Language Models — Introduction

A Large Language Model (LLM) is a type of AI model trained on massive amounts of text data — books, websites, code, research papers, and articles — to understand and generate human language. LLMs learn the statistical patterns of language so thoroughly that they can write essays, answer questions, summarise documents, translate languages, generate working code, and hold natural conversations.

LLMs are the foundation of tools like ChatGPT, Google Gemini, and Anthropic Claude. Under the hood, they are built using Transformer neural network architectures and trained using a technique called self-supervised learning — predicting the next token — on hundreds of billions of text tokens.

3.2 Key LLM Concepts

- **Input (Prompt):** the text instruction or question you send to the LLM. This is how you communicate what you want.
- **Output (Response):** the generated text returned by the LLM — an answer, summary, code snippet, or creative content.
- **Temperature (0–2):** controls randomness in the output. 0 = deterministic and factual (same answer every time); 2 = highly creative and unpredictable. 0.7 is the standard balanced default for most applications.
- **Tokens:** LLMs don't process whole words — they process tokens (chunks of characters). Approximately 1 token \approx 0.75 words. Token limits control the maximum length of input + output.
- **System Message:** a hidden instruction that sets the LLM's persona, behaviour, and constraints before the user speaks. For example: 'You are a data science tutor. Always explain simply.'
- **User Message:** the actual question or task from the user, sent after the system message.
- **Context Window:** the maximum number of tokens the LLM can 'see' at once — includes the full conversation history.

Temperature Value	Behaviour	Best Used For
0.0	Fully deterministic — same output every run	Factual Q&A, data extraction, code generation
0.3 – 0.5	Mostly consistent with slight variation	Summarisation, classification, structured tasks
0.7	Balanced — coherent yet varied (default)	General chatbots, question answering, tutoring
1.0 – 1.5	Creative and diverse responses	Brainstorming, creative writing, story generation
2.0	Highly random — sometimes incoherent	Experimental / artistic generation only

3.3 What is LangChain?

LangChain is an open-source Python framework that simplifies building applications powered by LLMs. Instead of writing raw HTTP API calls to OpenAI, Gemini, or other providers, LangChain provides a clean, standardised interface — so you can swap between different LLM providers with minimal code changes.

- **ChatGoogleGenerativeAI**: the LangChain class that connects to Google Gemini.
- **llm.invoke(messages)**: sends the formatted messages to the LLM and returns the response object.
- **response.content**: the text string extracted from the LLM's response.
- **Model-agnostic**: the same code structure works with OpenAI GPT, Google Gemini, Claude, LLaMA — only the import class changes.

💡 **Key Point:** In this guide we use Google Gemini 2.5 Flash — a fast, capable, and cost-efficient Google model accessible via the Google Generative AI API. Your `GOOGLE_API_KEY` from the `.env` file authenticates every call.

4. Code Example 1 — Basic LLM Call

4.1 What This Script Does

This is the simplest possible script to call an LLM from Python. It initialises the Google Gemini model via LangChain, constructs a message with a system prompt and a user question, sends it to the model, and prints the response. Think of this as your 'Hello World' for LLM development — every more complex application builds on exactly this pattern.

4.2 Concepts Introduced

- **load_dotenv()**: reads the `.env` file and loads `GOOGLE_API_KEY` into `os.environ` so the script can access it securely.
- **ChatGoogleGenerativeAI(...)**: initialises a connection to the Gemini model with configuration settings (model name, API key, temperature).
- **messages list**: a list of dictionaries, each with a 'role' and 'content'. This is how LangChain (and most LLM APIs) format conversations.
- **system role**: sets the AI's persona and behaviour rules before any user interaction.
- **user role**: contains the actual question or task from the human user.
- **llm.invoke(messages)**: sends all messages to the Gemini API and blocks until the response is received.
- **response.content**: the plain text string of the model's reply, extracted from the response object.

4.3 Full Code

```
1 llm_call.py
import os
from dotenv import load_dotenv
from langchain_google_genai import ChatGoogleGenerativeAI

load_dotenv()  # Reads .env and loads GOOGLE_API_KEY into os.environ

# — Initialise the Gemini model —
```



```

llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",          # model to use
    google_api_key=os.getenv("GOOGLE_API_KEY"), # key from .env
    temperature=0.7                  # 0 = factual, 2 = creative
)

# — Prepare messages —————
# system: defines the AI's persona/behaviour
# user:   the actual question
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user",   "content": "Give me a Joke on AI"}
]

# — Call the model and print the reply —————
response = llm.invoke(messages)
print(response.content)

```


4.4 How to Run

```

# Recommended: use poetry run (no manual venv activation needed)
poetry run python 1_llm_call.py

# Alternative: activate venv first, then run normally
# Windows:    .\.venv\Scripts\activate
# Mac/Linux: source .venv/bin/activate
python 1 llm call.py
► Output: Why did the AI go to therapy? It had too many deep issues! (Gemini's actual
response will vary)


```

 **Note:** The system message shapes every response that follows. Try changing it to 'You are a strict data scientist who only speaks in bullet points.' — all responses will reflect this persona until you change it.

5. Code Example 2 — Personalised Terminal Chatbot

5.1 What This Script Does

This script upgrades the basic one-shot LLM call into an interactive terminal chatbot. It runs a continuous loop, accepting user input from the keyboard, sending each message to Gemini, and printing the response — repeating until the user types 'exit' or 'quit'. This demonstrates how to build a simple conversational interface entirely in the terminal with no external dependencies beyond LangChain.

 **Note:** This version does not maintain conversation memory. Each message is sent as a fresh, independent API call. The model has no knowledge of what was said earlier in the same session. Implementing memory requires accumulating the full message history and sending it with every call.

5.2 New Concepts Introduced

- **while True:** creates an infinite loop that keeps the chatbot alive and waiting for input until explicitly stopped.

- **input('You: '):** pauses execution and waits for the user to type a message in the terminal, then returns it as a string.
- **user_input.lower():** converts the typed string to lowercase, so 'EXIT', 'Exit', and 'exit' all match the exit condition.
- **break:** immediately exits the while loop when the user types 'exit' or 'quit', ending the program.
- **f-string f'Bot: {response.content}':** formats the model's reply with a 'Bot: ' prefix for readable terminal output.

5.3 Full Code

```

2_llm_personalized_chatbot.py
import os
from dotenv import load_dotenv
from langchain_google_genai import ChatGoogleGenerativeAI

load_dotenv()

# — Initialise the model (same as File 1) —
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    google_api_key=os.getenv("GOOGLE_API_KEY"),
    temperature=0.7
)

print("--- Welcome to the Personalized Chatbot ---")
print("      (type 'exit' or 'quit' to end)      ")

# — Main chatbot loop —
while True:
    user_input = input("You: ")    # wait for user to type

    # Exit condition — stops the loop cleanly
    if user_input.lower() in ["exit", "quit"]:
        print("Goodbye!")
        break

    # Build messages fresh each turn (no memory between turns)
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": user_input}
    ]

    response = llm.invoke(messages)    # call Gemini
    print(f"Bot: {response.content}")    # print reply

```

5.4 How to Run

```

poetry run python 2_llm_personalized_chatbot.py

# Expected interaction in terminal:
# --- Welcome to the Personalized Chatbot ---
#      (type 'exit' or 'quit' to end)
# You: What is deep learning?
# Bot: Deep learning is a subset of machine learning...
# You: Give me an example
# Bot: Sure! One classic example is image recognition...

```

```
# You: exit
# Goodbye!
```

💡 Key Point: To add conversation memory: create a history list before the loop, append each user message and bot response to it after every turn, and send the full history list as 'messages' instead of just the latest message.

6. Streamlit & Code Example 3 — LLM Web UI

6.1 What is Streamlit?

Streamlit is an open-source Python library that lets you build interactive web applications for data science and AI using pure Python — no HTML, CSS, or JavaScript required. You write Python, and Streamlit automatically renders a polished, interactive browser-based UI. It is by far the fastest way to turn a Python script into a shareable web app.

Streamlit apps work on a reactive model — every time the user interacts with any widget (types in a box, clicks a button, moves a slider), Streamlit re-runs the entire Python script from top to bottom and refreshes the UI. This makes building interactive tools extremely simple.

6.2 Core Streamlit Components

Component	Code	What It Renders
Page title	<code>st.title('My App')</code>	Large H1 heading at the top of the page
Text display	<code>st.write('Hello World')</code>	Renders text, Markdown, data, or any Python object
Bold text	<code>st.write(**Bold**)</code>	Markdown formatting — bold , <i>italic</i> , etc.
Text input	<code>question = st.text_input('Label')</code>	Single-line text input box; stores user text in variable
Text area	<code>st.text_area('Label')</code>	Multi-line text input for longer content
Button	<code>if st.button('Click Me'):</code>	Clickable button — block runs when button is pressed
Subheader	<code>st.subheader('Section')</code>	Smaller heading for organising page sections
Spinner	<code>with st.spinner('Loading...'):</code>	Shows a loading spinner while a block runs
Success	<code>st.success('Done!')</code>	Green success message box
Error	<code>st.error('Something went wrong')</code>	Red error message box

💡 Key Point: Streamlit re-runs the entire script on every interaction. This means variables are reset on each run. Use `st.session_state` to persist values (like conversation history) across interactions.

6.3 What This Script Does

This script is the most complete of the three. It wraps the Gemini LLM in a browser-based web interface using Streamlit. The user opens a web page, types a question in a text input box, clicks 'Get Answer', and the LLM's response appears on the page — no terminal interaction needed. This is the foundation pattern for any production LLM web application.

6.4 New Concepts Introduced

- **import streamlit as st**: imports Streamlit with the standard 'st' alias used universally.
- **st.title()**: renders a large heading — the first thing the user sees on the page.
- **st.text_input('label')**: renders a text box. Whatever the user types is captured in the variable 'question'.
- **if st.button('Get Answer')**: renders a button and wraps the LLM call so it only fires when the button is clicked.
- **st.write('**Answer:**')**: renders bold text using Markdown formatting — double asterisks make text bold.
- **st.write(llm.invoke(messages).content)**: chains the LLM call and display in one line — calls Gemini and renders the result.

6.5 Full Code

```
3 llm_with_ui.py
import os
from dotenv import load_dotenv
from langchain_google_genai import ChatGoogleGenerativeAI
import streamlit as st          # import Streamlit

load_dotenv()      # Load GOOGLE_API_KEY from .env

# — Initialise Gemini model —————
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    google_api_key=os.getenv("GOOGLE_API_KEY"),
    temperature=0.7,
)

# — Streamlit UI layout —————
st.title("Google Gemini LLM with Streamlit")

# Text input box — question variable holds whatever the user types
question = st.text_input("Enter your question:")

# Button — entire indented block runs only when button is clicked
if st.button("Get Answer"):
    messages = [
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": question}
    ]
    st.write('**Answer:**')          # bold heading via Markdown
    st.write(llm.invoke(messages).content)  # call LLM and show reply
```

6.6 How to Run — IMPORTANT


Streamlit scripts use a different run command. Do NOT use 'python filename.py' — this will not start the web server. Always use 'streamlit run'.

```
# Recommended: use poetry run
poetry run streamlit run 3_llm_with_ui.py

# Alternative: with venv activated
streamlit run 3_llm_with_ui.py
```

```
# Streamlit automatically opens your browser at:
# http://localhost:8501

# To stop the server: press Ctrl+C in the terminal
```

 **Note:** If the browser does not open automatically, manually navigate to <http://localhost:8501> in any browser. The Streamlit app runs as a local server — it is only accessible on your own machine unless you deploy it to the cloud.

7. Complete Setup Checklist

Use this checklist to verify your environment is fully configured before running the scripts:

#	Step	Command / Action	Done?
1	Install Python 3.12	python.org/downloads — tick 'Add to PATH'	<input type="checkbox"/>
2	Install VSCode	code.visualstudio.com	<input type="checkbox"/>
3	Create project folder & open in VSCode	File → Open Folder	<input type="checkbox"/>
4	Create virtual environment	<code>py -3.12 -m venv .venv</code> (Windows)	<input type="checkbox"/>
5	Activate virtual environment	<code>.\venv\Scripts\activate</code> (Windows)	<input type="checkbox"/>
6	Install Poetry	<code>pip install poetry</code>	<input type="checkbox"/>
7	Initialise Poetry project	<code>poetry init</code>	<input type="checkbox"/>
8	Create .env file with API key	<code>GOOGLE_API_KEY=your_key</code>	<input type="checkbox"/>
9	Get Google Gemini API key	aistudio.google.com/app/apikey	<input type="checkbox"/>
10	Install all dependencies	<code>poetry add langchain==0.3.27 python-dotenv==1.1.1 ...</code>	<input type="checkbox"/>
11	Add .env to .gitignore	<code>echo .env >> .gitignore</code>	<input type="checkbox"/>
12	Run File 1 — basic LLM call	<code>poetry run python 1_llm_call.py</code>	<input type="checkbox"/>
13	Run File 2 — terminal chatbot	<code>poetry run python 2_llm_personalized_chatbot.py</code>	<input type="checkbox"/>
14	Run File 3 — Streamlit web UI	<code>poetry run streamlit run 3_llm_with_ui.py</code>	<input type="checkbox"/>

8. Troubleshooting

Error / Issue	Likely Cause	Fix
'py' is not recognized	Python not added to PATH	Reinstall Python 3.12 — tick 'Add Python to PATH'
Execution Policy error (Windows)	PowerShell security restriction	Run: <code>Set-ExecutionPolicy -ExecutionPolicy Bypass -Scope Process</code>
'poetry' not found after install	pip installed in wrong environment	Ensure .venv is activated first, then run <code>pip install poetry</code>

GOOGLE_API_KEY returns None	.env file missing or load_dotenv not called	Create .env in project root; call load_dotenv() before os.getenv()
ImportError: No module named...	Package not installed or wrong environment	Run: poetry install OR poetry add <package-name>
Streamlit not opening browser	Browser didn't auto-open	Manually go to http://localhost:8501
'streamlit' not recognized as command	Script run with python instead of streamlit	Use: poetry run streamlit run 3_llm_with_ui.py
API quota / rate limit error	Too many Gemini API calls in a short time	Wait a few minutes or check quota at console.cloud.google.com
poetry.lock conflict on install	Incompatible package versions	Run: poetry update to resolve and regenerate lock file