OPEN-SOURCE LLM + WHATSAPP = YOUR OWN AI ASSISTANT

Qwen2.5

PREM KUMAR R
Neural Bytes with Prem

Contents

What Is Ollama?

Ollama is a tool that helps you **run open-source Large Language Models (LLMs)** easily on your local machine or server — just like you would run Docker containers.

Think of it like:

Docker → for software containers
Ollama → for LLMs (like Qwen, Mistral, LLaMA, etc.)

Why Do We Use Ollama?

Without Ollama, you'd need to:

- Download gigabytes of model weights manually
- Set up complex inference environments (Python, CUDA, PyTorch, etc.)
- Write boilerplate code to serve the model

With Ollama, you can instead:

```
ollama pull qwen:7b       # Download the model
ollama run qwen:7b        # Run it immediately
```

It simplifies **model downloading, optimization, serving, and inference**.

What Is qwen:7b?

Qwen-7B is:

- A **7-billion parameter LLM** from Alibaba Group
- Open-sourced under a permissive license (Qianwen project)
- Supports **multilingual reasoning** (English, Chinese, Arabic, etc.)
- Comparable to LLaMA 2 or Mistral in performance

Qwen is especially known for **strong reasoning and instruction-following**, and is optimized for real-world use.

Now Let's Break Down the Code : Ollama & Qwen:7b

```
curl -fsSL https://ollama.com/install.sh | sh
```

- `curl`: downloads a shell script
- `-fsSL`: fail silently, show errors, follow redirects
- `| sh`: pipes the script into the shell to **auto-install Ollama**

→ This line **installs the Ollama CLI tool**.

```
ollama serve &
```

- Starts the **Ollama backend service** (a local LLM server)
- The `&` at the end runs it in the **background**, so your terminal is still usable

→ This is needed for Ollama to accept LLM execution requests.

```
ollama pull qwen:7b
```

- Downloads the **Qwen 7B model** to your system
- Prepares it for inference so it can respond to prompts

→ It's similar to:

```
docker pull qwen:7b
```

Qwen_api.py: Purpose

This script:

- Creates a simple **Flask-based API server**
- Accepts a **POST request** with a prompt (text message)
- Uses `ollama run qwen:7b` to generate a response
- Returns the LLM's reply as JSON

Line-by-Line Breakdown

```
from flask import Flask, request, jsonify
```

- **Flask**: lightweight web framework to build APIs
- `request`: lets you access incoming HTTP request data (e.g., JSON body)
- `jsonify`: helps format Python data (e.g., dictionaries) into JSON HTTP responses

---

```
app = Flask(__name__)
```

- Initializes the Flask app instance
- `__name__` helps Flask locate resources (e.g., templates/static files, though not used here)

```
@app.route("/generate", methods=["POST"])
```

- This is a **route decorator** that maps the URL `/generate` to the function below
- It **only allows POST requests** (not GET/PUT/DELETE)

Example of a valid request:

```
POST /generate
{ "prompt": "What is the capital of India?" }
```

```
def generate():
    prompt = request.json.get("prompt")
```

- This function runs whenever a request hits `/generate`
- It extracts the `"prompt"` value from the JSON body of the request

Example:

```
{"prompt": "Tell me a joke"}
→ prompt = "Tell me a joke"
```

```
    result = subprocess.run(["ollama", "run", "qwen:7b", prompt],
capture_output=True, text=True)
```

- This runs the following **shell command** via Python:

```
ollama run qwen:7b "Tell me a joke"
```

- `subprocess.run(...)` lets you run shell commands from Python
- `capture_output=True` captures the response (stdout) instead of printing it
- `text=True` ensures the output is returned as a string (not bytes)

```
    return jsonify({"response": result.stdout.strip()})
```

- Sends a JSON response back to the client
- The `.strip()` removes any extra newlines/spaces around the LLM's response

Output example:

```
{"response": "The capital of India is New Delhi."}
```

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

- This tells Python to **start the Flask server** only if this file is run directly
- `host="0.0.0.0"` allows the server to accept connections from any IP (important for RunPod)
- `port=5000` defines the HTTP port the API will listen on

You can then POST to:

```
http://<your-server-ip>:5000/generate
```

Summary Flow - qwen_api.py

1. User sends a prompt to `/generate`
2. Flask receives it, extracts `"prompt"`
3. Python runs: `ollama run qwen:7b "<prompt>"`
4. LLM responds
5. Response is returned as JSON

Why do we use cloudflared even though qwen_api.py is already running?

You're absolutely right that `qwen_api.py` launches a web server on port `5000`, **but**:

That server is **only accessible inside the RunPod container**, not to the outside world.

So unless you expose it, **Twilio, your browser, or any external client can't access it**.

The Problem

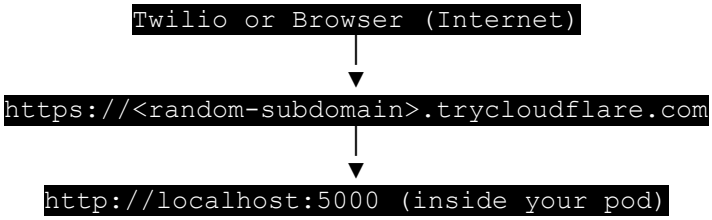
Flask's `app.run(host="0.0.0.0", port=5000)`:

- ☑ Opens an HTTP server **inside** your GPU pod
- ❌ Does *not* expose a public IP or DNS endpoint
- ❌ Not accessible over internet unless you:
  - Use RunPod's HTTP Service manually
  - Or forward traffic using something like `cloudflared`

What cloudflared Does

cloudflared creates a **publicly accessible HTTPS tunnel** from the outside world to your **local Flask server** (running inside the pod on port 5000).

Think of it like:



Why We Prefer cloudflared for this

- Instant public URL (HTTPS secured)
- No need to open firewall ports manually
- Great for testing with tools like Twilio or WhatsApp
- No Cloudflare login needed (for "quick tunnels")
- Works even when you're on a free or private GPU pod

Breakdown of Each Line: Cloudflared

```
wget https://github.com/cloudflare/cloudflared/releases/latest/download/cloudflared-linux-amd64
```

➡️ Downloads the Linux binary of `cloudflared`

```
mv cloudflared-linux-amd64 cloudflared
chmod +x cloudflared
```

➡️ Renames and makes it executable

```
./cloudflared tunnel --url http://localhost:5000
```

➡️ Creates a public tunnel to your **Flask server running on port 5000**

Example output:

```
Your quick Tunnel has been created! Visit it at:
https://blue-fox-quick.trycloudflare.com
```

What Happens Without This?

If you skip this step:

- Your server runs fine **inside RunPod**
- But tools like **Twilio**, **Postman**, or your browser will get:

```
ERR_CONNECTION_TIMED_OUT
```

or

```
Connection refused
```

Because there's no path to reach `localhost:5000` from the outside.

Summary

Concept	What it does
`qwen_api.py`	Runs a Flask API inside the pod
`cloudflared`	Creates a public tunnel to that internal server
Why needed	External services (like Twilio) can't access pod-local ports directly

What Is whatsapp_bot.py Doing?

This script is a **Flask-based webhook** that:

1. Receives **incoming WhatsApp messages** from Twilio
2. Forwards them to your **Qwen2.5 LLM API**
3. Sends the LLM's response **back to the user on WhatsApp**

Line-by-Line Breakdown : WhatsApp_Bot.py

```
from flask import Flask, request, Response
import requests
```

- We import:
  - `Flask`: for building a web server
  - `request`: to read incoming POST data from Twilio
  - `Response`: to return TwiML (Twilio's XML message format)
  - `requests`: to call the LLM API

```
app = Flask(__name__)
LLM_API_URL = "https://<your-cloudflared-url>/generate"
```

- Create the Flask app
- Set the URL of your Qwen2.5 API exposed via `cloudflared` — this is where your webhook will **forward user prompts**

```
@app.route("/whatsapp", methods=["POST"])
```

```
def whatsapp_reply():
```

- Define a route called `/whatsapp`
- Twilio will POST messages to this route

```
    incoming_msg = request.form.get("Body", "").strip()
```

- Extract the user's WhatsApp message from the form payload (comes in as `Body`)

```
    response = requests.post(LLM_API_URL, json={"prompt": incoming_msg})
    llm_reply = response.json().get("response", "Sorry, I didn't understand
that.")
```

- Forward the user's message to the Qwen2.5 API
- Parse the reply from the LLM
- Use a fallback message if the response is missing

```
    return Response(f"""<?xml version="1.0" encoding="UTF-8"?>
<Response>
    <Message>{llm_reply}</Message>
</Response>""", mimetype="application/xml")
```

- Return a TwiML response that Twilio uses to send a WhatsApp reply back to the user
- Twilio requires responses in **XML**, not JSON

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5001)
```

- Starts the server on port 5001, accessible from inside the pod

---

Summary of What this Bot Does:

Source	Action
Twilio	Sends WhatsApp message → Flask
Flask Server	Extracts text → sends to Qwen2.5
Qwen2.5 API	Returns intelligent response
Flask Server	Sends it back to user via Twilio

Why Expose with Cloudflared Again?

```
./cloudflared tunnel --url http://localhost:5001
```

Just like we exposed `qwen_api.py` on port `5000`, we must now expose this second Flask app on port `5001`.

Why?

- **Twilio Sandbox** is a cloud service
- It needs a **publicly accessible HTTPS endpoint** to deliver messages to
- By default, your Flask app is only reachable inside RunPod
- `cloudflared` makes it public by generating a URL like:

```
https://improve-perspective-wright-ts.trycloudflare.com/whatsapp
```

What Happens Once Live

```
 [User's WhatsApp] → [Twilio] → [Your webhook (/whatsapp)] → [Qwen2.5 API] → [Your webhook] → [Twilio] → [User]
```

That's the **real-time round trip**.

Why This Setup Is Important

Feature	Why It Matters
Cloudflared on 5001	Required for Twilio to talk to your bot
Flask webhook endpoint	Receives and processes WhatsApp messages
Qwen2.5 API call	Makes the chatbot intelligent
XML TwiML response	Required format for Twilio replies