

How can we set up a chatbot on WhatsApp using Twilio API?

Below is a **clear, step-by-step guide** for setting up a WhatsApp chatbot using **Twilio's WhatsApp API**, whether you're building something simple or integrating with a large language model (like ChatGPT or Vertex AI).

Text

Markdown

Screenshot

PDF

Word

HTML

JSON

Excel

CSV

Hide

✓ 1. Prerequisites

Before you start, make sure you have:

✓ A Twilio account

Sign up at <https://www.twilio.com/> if you don't have one.

✓ A verified WhatsApp number (sandbox or approved business)

You can start with Twilio's **WhatsApp Sandbox** for free.

✓ A backend environment

Node.js, Python, Java, or any stack.

Below, I'll show examples in **Node.js** and **Python**.

🚀 2. Activate Twilio WhatsApp Sandbox

1. Go to **Twilio Console** → **Messaging** → **Try it out** → **WhatsApp Sandbox**.
2. You will see a sandbox number and a **join code**.
3. On your phone, send the join code to the sandbox number via WhatsApp.

This pairs your personal WhatsApp with your Twilio sandbox.

⚙️ 3. Set Up a Webhook URL

WhatsApp messages are delivered to your server through a **webhook**.

You need a public HTTPS URL. You can use:

Tool	Purpose
ngrok	Expose localhost → https://random.ngrok.io/webhook ↗
Render / Railway / Vercel	Deploy backend easily

🤖 4. Build the Chatbot Backend

Option A — Node.js Example (Express)

★ Install dependencies:

bash



```
npm init -y
npm install express twilio body-parser
```

★ index.js:

javascript



```
const express = require('express');
const bodyParser = require('body-parser');
const { MessagingResponse } = require('twilio').twiml;

const app = express();
app.use(bodyParser.urlencoded({ extended: false }));

app.post('/whatsapp', (req, res) => {
  const twiml = new MessagingResponse();

  const msg = twiml.message(`You said: ${req.body.Body}`);

  res.writeHead(200, { 'Content-Type': 'text/xml' });
  res.end(twiml.toString());
});
```

```
app.listen(3000, () => console.log('Server running on port 3000'));
```

★ Run locally:

bash

Copy code

```
node index.js  
ngrok http 3000
```

Copy the <https://xxxx.ngrok.io> URL.

Option B — Python Example (Flask)

★ Install dependencies:

bash

Copy

```
pip install flask twilio
```

★ app.py:

python

Copy

```
from flask import Flask, request  
from twilio.twiml.messaging_response import MessagingResponse  
  
app = Flask(__name__)  
  
@app.route("/whatsapp", methods=["POST"])  
def reply():  
    incoming_msg = request.form.get("Body")  
    resp = MessagingResponse()  
    resp.message(f"You said: {incoming_msg}")  
    return str(resp)  
  
if __name__ == "__main__":  
    app.run(port=3000)
```

★ Run:

bash

 Copy code

```
python app.py  
ngrok http 3000
```

🔗 5. Connect Twilio Webhook

1. Go to your Twilio Console → WhatsApp Sandbox.
2. Find "When a message comes in".
3. Paste your webhook URL:

arduino



<https://xxxx.ngrok.io/whatsapp>

Now Twilio will POST to this URL when you receive WhatsApp messages.

🧠 6. (Optional) Add AI Chatbot Logic

You can integrate OpenAI, Vertex AI, LangChain, or your own LLM.

Example with OpenAI (Node.js):

javascript

 Copy code

```
const openai = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });

app.post('/whatsapp', async (req, res) => {
  const userText = req.body.Body;
  const aiReply = await openai.chat.completions.create({
    model: "gpt-4o-mini",
    messages: [{ role: "user", content: userText }]
  });

  const twiml = new MessagingResponse();
  twiml.message(aiReply.choices[0].message.content);
```

```
res.type('text/xml').send(twiml.toString());  
});
```

📌 Deployment Options

You can deploy the chatbot on:

- **Render.com** (free & easy)
- **Railway.app**
- **Google Cloud Run**
- **Heroku** (paid)
- **Vercel** (serverless)

All support Node.js and Python.

🎯 Summary

Step	Description
1	Sign up for Twilio
2	Activate WhatsApp Sandbox
3	Create webhook server
4	Write chatbot logic
5	Connect webhook to Twilio
6	(Optional) Integrate AI

You're done — your WhatsApp chatbot is live!

💬 Want me to help you build your exact chatbot?

I can help you:

- Deploy it on Render or Cloud Run

- Add AI reasoning or memory
- Connect to DynamoDB, Firestore, PostgreSQL
- Build a menu-based WhatsApp bot
- Build a customer support bot for your startup

Just tell me what your use case is!



I'm using Python,

Great — here is a **clean, working Python + Flask + Ngrok setup specifically for WhatsApp chatbot using Twilio API.**

Follow these steps exactly and it will work end-to-end.

✓ 1. Install Dependencies

Make sure you have Python 3.8+.

bash



```
pip install flask twilio
```

✓ 2. Create app.py

This is a minimal Flask app that reads WhatsApp messages and replies through Twilio.

python



```
from flask import Flask, request
from twilio.twiml.messaging_response import MessagingResponse

app = Flask(__name__)

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

```
def whatsapp_reply():
    incoming_msg = request.form.get("Body")

    resp = MessagingResponse()
    resp.message(f"You said: {incoming_msg}")

    return str(resp)

if __name__ == "__main__":
    app.run(port=5000, debug=True)
```

✓ 3. Run the Flask Server

bash



python app.py

This starts your backend at:

arduino



<http://localhost:5000>

✓ 4. Start ngrok

Expose your local server to the internet:

bash



ngrok http 5000

You will see something like:

nginx



Forwarding https://1234abcd.ngrok.io → http://localhost:5000

Copy the HTTPS URL.

✓ 5. Set Twilio Webhook (WhatsApp Sandbox)

Go to:

👉 Twilio Console → Messaging → WhatsApp Sandbox

Find:

"When a message comes in"

Paste your webhook URL:

arduino



<https://1234abcd.ngrok.io/whatsapp>

Save.

✓ 6. Test the Bot

On your phone, open WhatsApp and message your Twilio sandbox number

nginx



hi

You should receive:

yaml



Copy code

You said: hi

Your WhatsApp chatbot is officially working!

💡 Add AI (Optional)

If you want your bot to respond intelligently using OpenAI:

python

```
from flask import Flask, request
from twilio.twiml.messaging_response import MessagingResponse
from openai import OpenAI
import os

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
app = Flask(__name__)

@app.route("/whatsapp", methods=["POST"])
def whatsapp_reply():
    user_msg = request.form.get("Body")

    completion = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": user_msg}]
    )

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

    resp = MessagingResponse()
    resp.message(ai_reply)

    return str(resp)
```

? Want me to help debug your setup?

I can help you fix:

- Webhook not firing
- Twilio 11200 / 30008 errors
- Ngrok not forwarding
- Flask server issues
- Environment variables / OpenAI API key
- Deployment (Render, Cloud Run, Railway)

Just tell me what you have so far and where you're stuck!



How to create a state management in chatbot? Here's my current code

```
def handle_state(self, from_number: str, conv_state: dict, message: str, media_url: Optional[str]) -> str:
    state = conv_state['state']
    data = conv_state.get('data', {})

    # Role selection
    if state == 'awaiting_role_selection':
        if message.strip() == '1':
            self.store.create_user(from_number,
                                   return
        self.start_farmer_registration(from_number)
        elif message.strip() == '2':
            self.store.create_user(from_number,
                                   'farm_owner')
            return
        self.start_owner_registration(from_number)
        else:
            return "Please reply with 1 (for Farmer) or 2 (for Farm Owner)"
    elif state == 'farmer_reg_name':
        return self.handle_farmer_name(from_number,
                                       message)
    elif state == 'farmer_reg_location':
        return self.handle_farmer_location(from_number,
                                           message)
    elif state == 'farmer_reg_id':
        return self.handle_farmer_id(from_number,
                                     media_url)

    elif state == 'farmer_update_menu':
        return self.handle_update_menu(from_number,
                                       message)
    elif state == 'farmer_update_work_type':
        return
    self.handle_work_type_update(from_number, message)
```

```
elif state == 'farmer_update_pay_rate':  
    return self.handle_pay_rate_update(from_number,  
message)  
elif state == 'farmer_update_distance':  
    return self.handle_distance_update(from_number,  
message)  
elif state == 'farmer_update_hours':  
    return self.handle_hours_update(from_  
message)  
elif state == 'farmer_pref_work_type':  
    return self.handle_work_type(from_nur  
message)  
elif state == 'farmer_pref_pay_rate':  
    return self.handle_pay_rate(from_numk  
message)  
elif state == 'farmer_pref_location':  
    return self.handle_pref_location(from_i  
message)  
elif state == 'farmer_pref_actual_location':  
    return  
self.handle_actual_location_update(from_nun  
message)  
elif state == 'farmer_pref_hours':  
    return self.handle_hours(from_number,  
  
elif state == 'viewing_jobs':  
    return self.handle_job_selection(from_number,  
message, data)  
elif state == 'selecting_from_recommendations':  
    return  
self.handle_job_selection_from_list(from_number,  
message, data)  
elif state == 'reviewing_recommendation':  
    return  
self.handle_recommendation_action(from_number,  
message, data)  
elif state == 'job_details_view':  
    return self.handle_job_application(from_number,  
message, data)  
elif state == 'job_action':
```

```

        return self.handle_job_action(from_number,
message, data)

        # Direct messaging
        elif state == 'chatting':
            return self.handle_chat_message(from_number,
message, data)

        return "I didn't understand that. Please try"
        type 'menu' for main menu."
    
```

You're **very close** to a clean state-managed WhatsApp chatbot — you already have a state machine, but it needs to be structured so that:

- 1. Every state transition is predictable**
- 2. Your state is saved after every message**
- 3. Your message handlers update the state**
- 4. Your code is maintainable (not one giant if-else block)**

Below I'll show you:

✓ 1. What You Currently Have

Your function is already acting as a **state dispatcher**:

python

 Copy code

```

def handle_state(self, from_number, conv_state, message, media_url):
    state = conv_state['state']
    
```

This is correct — but it's missing:

- **A consistent place to update the next state**
 - **A centralized state table**
 - **Reduced repetition**
 - **Easy addition of new states**
-

✓ 2. Ideal State Machine Pattern

A clean architecture for chatbot state:

pgsql

 Copy code

```
User sends message →
Load user_state →
Pass to dispatcher →
Run handler →
Return reply + next_state →
Save state →
Send reply
```

Meaning **each handler must return the next state**, not just a response message.

✓ 3. Recommended Refactor: Use a STATE MACHINE

Instead of long `if/elif`, use a dictionary:

python

 Copy

```
STATE_HANDLERS = {
    'awaiting_role_selection': self.handle_role_selection,
    'farmer_reg_name': self.handle_farmer_name,
    'farmer_reg_location': self.handle_farmer_location,
    'farmer_reg_id': self.handle_farmer_id,
    'farmer_update_menu': self.handle_update_menu,
    ...
}
```

Then your dispatcher becomes:

python

 Copy code

```
def handle_state(self, from_number, conv_state, message, media_url):
    state = conv_state['state']
    handler = self.STATE_HANDLERS.get(state)

    if handler:
        return handler(from_number, message, media_url, conv_state)
```

```
return "I didn't understand that. Type 'menu' to restart."
```

Much cleaner.

✓ 4. Example Handler Pattern

Each handler should:

- 1 Process input
- 2 Decide the next state
- 3 Save that next state
- 4 Return the bot message

Example:

python



```
def handle_farmer_name(self, from_number, message, *_):  
    # save the name  
    self.store.update_user(from_number, {"name": message})  
  
    # set next state  
    self.store.update_state(from_number, "farmer_reg_location")  
  
    return "Thanks! Now, please send your location."
```

That's the correct pattern.

✓ 5. Your Current Code's Main Weakness

Right now each handler:

- ✗ does not update the next state explicitly
- ✗ returns only a string, not a transition
- ✗ forces all state logic inside the dispatcher

This makes it hard to maintain and scale.

✓ 6. Recommended State Manager Structure

A. Dispatcher

python



```
def handle_state(self, from_number, conv_state, message, media_url):
    state = conv_state.get('state')
    handler = self.STATE_HANDLERS.get(state)

    if not handler:
        return "State error. Type 'menu' to restart."

    return handler(from_number, message, media_url, conv_state)
```

B. Example state map

python



```
STATE_HANDLERS = {
    'awaiting_role_selection': self.handle_role_selection,
    'farmer_reg_name': self.handle_farmer_name,
    'farmer_reg_location': self.handle_farmer_location,
    'farmer_reg_id': self.handle_farmer_id,
    'farmer_update_menu': self.handle_update_menu,
    # ... add more here
}
```

✓ 7. Example Complete Handler

python

Copy code

```
def handle_role_selection(self, from_number, message, *_):
    msg = message.strip()
```

```

if msg == '1':
    self.store.create_user(from_number, 'farmer')
    self.store.update_state(from_number, 'farmer_reg_name')
    return "Great! What is your name?"

elif msg == '2':
    self.store.create_user(from_number, 'farm_owner')
    self.store.update_state(from_number, 'owner_reg_name')
    return "Great! What is your name?"

return "Invalid input. Reply 1 for Farmer or 2 for Farm"

```

✓ 8. How Your Database Should Store State

Each user should have a record:

json



```
{
  "phone": "+123456789",
  "state": "farmer_reg_location",
  "data": {
    "name": "John Doe"
  }
}
```

📌 9. Summary of Best Practices

What	Why
Use a handler map	Eliminates giant if/elif block
Each handler sets next state	Prevents conversation from derailing
Save state after each message	Persist conversation
Store temporary form data in data	Clean separation of state vs payload

What

Keep dispatcher thin

Why

Easier to extend

🎯 If you want, I can:

- ✓ Refactor your entire state system into a clean class
- ✓ Add automatic state transitions (finite-state machine)
- ✓ Add conversation restart logic
- ✓ Add session expiration / timeout
- ✓ Build a diagram of your chatbot flow

Would you like me to generate the **full improved code structure** based on handlers?



How to write data from whatsapp messages fr
into JSON data store using CRUD operations?

Below is a **clean, practical way** to store WhatsApp user data in a **JSON “database”** using **CRUD operations** — perfect for small chatbots or prototypes.

I will give you:

1. A JSON datastore structure
2. A Python class with CRUD methods
3. How to read/write user messages into JSON
4. How to integrate it with your state machine

✓ 1. JSON Data Store Structure

Create a file:

pgsql

 Copy code**data_store.json**

Example structure:

json



```
{
  "users": {
    "+123456789": {
      "role": "farmer",
      "state": "farmer_reg_location",
      "data": {
        "name": "John",
        "location": "Kenya"
      }
    }
  }
}
```

✓ 2. Create a JSON Data Store Manager (CRU)

python

 Copy code

```
import json
import os

class JSONStore:
    def __init__(self, file_path="data_store.json"):
        self.file_path = file_path

        # Create file if missing
        if not os.path.exists(self.file_path):
            with open(self.file_path, "w") as f:
                json.dump({"users": {}}, f)

    def _load(self):
        with open(self.file_path, "r") as f:
```

```
        return json.load(f)

    def _save(self, data):
        with open(self.file_path, "w") as f:
            json.dump(data, f, indent=4)
```

◆ A. Create user

python



```
def create_user(self, phone, role):
    data = self._load()

    data["users"][phone] = {
        "role": role,
        "state": "awaiting_next_input",
        "data": {}
    }

    self._save(data)
    return data["users"][phone]
```

◆ B. Read user

python



Copy code

```
def get_user(self, phone):
    data = self._load()
    return data["users"].get(phone)
```

◆ C. Update user fields

python



Copy code

```
def update_user(self, phone, updates: dict):
    data = self._load()

    if phone not in data["users"]:
        return None

    data["users"][phone]["data"].update(updates)

    self._save(data)
    return data["users"][phone]
```

◆ D. Update conversation state

python



```
def update_state(self, phone, state):
    data = self._load()

    if phone not in data["users"]:
        return None

    data["users"][phone]["state"] = state

    self._save(data)
    return state
```

◆ E. Delete user

python



```
def delete_user(self, phone):
    data = self._load()

    if phone in data["users"]:
        del data["users"][phone]
        self._save(data)
```

```
return True
```

```
return False
```

3. USING CRUD IN YOUR CHATBOT

Example: When a new user sends "1" for farmer registration:

python



```
if user_message == "1":  
    self.store.create_user(phone, "farmer")  
    self.store.update_state(phone, "farmer_reg_name")  
  
    return "Great! What is your name?"
```

Example: Save name message into JSON

python



```
def handle_farmer_name(self, phone, message, *_):  
    self.store.update_user(phone, {"name": message})  
    self.store.update_state(phone, "farmer_reg_location")  
  
    return "Thanks! Now send your location."
```

Example: Save job preferences

python



Copy code

```
self.store.update_user(phone, {  
    "preferred_work_type": message  
})  
self.store.update_state(phone, "farmer_pref_pay_rate")
```

✓ 4. Example Read User Data

python

Copy code

```
user = self.store.get_user(phone)
role = user["role"]
state = user["state"]
info = user["data"]
```

🔥 FULL FLOW EXAMPLE

User sends:

nginx

Copy

John

Your handler:

python

Copy

```
def handle_farmer_name(self, phone, message, *_):
    self.store.update_user(phone, {"name": message})
    self.store.update_state(phone, "farmer_reg_location")
    return "Great, John! Now share your location."
```

This writes to JSON:

json

Copy code

```
{
  "users": {
    "+123456789": {
      "role": "farmer",
      "state": "farmer_reg_location",
      "data": {
        "name": "John"
      }
    }
  }
}
```

```
}
```

🚀 Want to Upgrade Later?

When your system grows, you can easily migrate this logic to:

- **SQLite / PostgreSQL**
- **Firebase Realtime Database**
- **MongoDB**
- **DynamoDB**

Your CRUD interface will stay the same — we only replace the backend.

🎯 If you'd like, I can generate:

- ✓ A fully working Flask + Twilio + JSON datastore bot template
- ✓ Your complete state machine refactored into a clean architecture
- ✓ A diagram showing user flow and state transitions
- ✓ A migration path from JSON → real database

Would you like the **full production-ready bot structure?**



I need to create a chatbot help user create a job profile and match top 5 jobs recommended for users, and save the match if users choose and apply to it. Guide me create this workflow using framework that we walkthrough before on WhatsApp

Below is a **complete end-to-end workflow design** for your WhatsApp chatbot that:

- ✓ Helps users create a job profile

- ✓ Matches them with top 5 recommended jobs
- ✓ Lets them view details, choose, and apply
- ✓ Saves job applications into your JSON CRUD datastore
- ✓ Integrates cleanly with the state-machine architecture we discussed

This gives you a **production-ready flow**, clear state transitions, and all data

HIGH-LEVEL WORKFLOW

A good job-matching bot has 4 main phases:

1 User Registration (Build Job Profile)

Collect everything needed for job matching:

1. Name
 2. Location
 3. Work type (full-time / part-time / skilled type)
 4. Preferred pay rate
 5. Maximum travel distance
 6. Working hours preference
 7. Actual current location (optional)
-

2 Job Matching

You run a matching algorithm:

- Filter jobs by location
- Filter by work type
- Sort by pay rate match
- Sort by distance
- Sort by hours match
- Take **top 5**

Return results in WhatsApp like:

bash

 Copy code

Here are your top 5 job recommendations:

1. Harvest Worker – \$12/hr – 3km away
2. Irrigation Assistant – \$15/hr – 5km
3. Seed Planter – \$10/hr – 2km
- ...

Reply with the job number to view details.

3 Job Details → Apply

When user selects a job:

- Show full details
- Ask whether they want to apply

vbnet



You selected Harvest Worker.

Pay: \$12/hr

Hours: 6hr/day

Location: Nairobi

Description: Assist with crop harvesting.

Reply:

1. Apply
 2. Go back to recommendations
-

4 Save Application

When user applies:

- Save to data store:
 - user_phone
 - job_id

- timestamp
 - application_status
 - Notify farm owner (optional)
 - Return success message
-

🧠 STATE MACHINE FOR ENTIRE WORKFLOW

Below is the recommended state flow:

powershell



```
START
↓
awaiting_role_selection
↓
farmer_reg_name
↓
farmer_reg_location
↓
farmer_pref_work_type
↓
farmer_pref_pay_rate
↓
farmer_pref_distance
↓
farmer_pref_hours
↓
viewing_jobs
↓
selecting_from_recommendations
↓
job_details_view
↓
job_action
↓
job_applied
```

Your dispatcher will route based on state like this:

python

```
STATE_HANDLERS = {
    'awaiting_role_selection': self.handle_role_selection,
    'farmer_reg_name': self.handle_farmer_name,
    'farmer_reg_location': self.handle_farmer_location,
    'farmer_pref_work_type': self.handle_work_type,
    'farmer_pref_pay_rate': self.handle_pay_rate,
    'farmer_pref_distance': self.handle_distance,
    'farmer_pref_hours': self.handle_hours,
    'viewing_jobs': self.handle_job_list,
    'selecting_from_recommendations': self.handle_job_select,
    'job_details_view': self.handle_job_details,
    'job_action': self.handle_job_action,
}
```

📦 JSON DATASTORE STRUCTURES

✓ User Profile

json

5

```
{
  "users": {
    "+123456": {
      "role": "farmer",
      "state": "viewing_jobs",
      "data": {
        "name": "John",
        "location": "Nakuru",
        "preferred_work_type": "Harvest",
        "preferred_pay_rate": 12,
        "distance": 10,
        "preferred_hours": 6
      }
    }
  }
}
```

✓ Saved Jobs Data

json

Copy code

```
{
  "jobs": {
    "job_001": {
      "title": "Harvest Worker",
      "pay_rate": 12,
      "location": "Nakuru",
      "distance_km": 3,
      "hours": 6,
      "required_skill": "harvest"
    }
  }
}
```

✓ Job Applications

json

Copy code

```
{
  "applications": [
    {
      "user": "+123456",
      "job_id": "job_001",
      "status": "applied",
      "timestamp": "2025-01-22T10:02:15"
    }
  ]
}
```



LOGIC: HOW TO MATCH TOP 5 JOBS

python

Copy code

```

def match_jobs(self, user_profile):
    jobs = self.store.get_all_jobs()
    user = user_profile['data']

    scored_jobs = []

    for job_id, job in jobs.items():
        score = 0

        if job["required_skill"].lower() == user["preferred_"]:
            score += 50

        score += max(0, 20 - abs(job["pay_rate"] - user["pre"])
        score += max(0, 10 - abs(job["hours"] - user["prefer"
        score += max(0, user["distance"] - job["distance_km"])

    scored_jobs.append((score, job_id, job))

    # Sort by score (highest first)
    scored_jobs.sort(reverse=True)

    # Return top 5
    return scored_jobs[:5]

```

WHATSAPP MESSAGES FLOW

1 After registration:

arduino

 Copy code

Your profile is ready!

Finding the best jobs for you...

2 Show job list:

bash

 Copy code

Here are your top 5 job matches:

1. Harvest Worker - 3km - \$12/hr
2. Seed Planter - 2km - \$10/hr
3. Irrigation Assistant - 5km - \$15/hr

Reply with a number to see details.

Set state:

python



```
self.store.update_state(phone, "selecting_from_recommendatio
```

🔍 When user chooses job "1"

vbnet



You selected: Harvest Worker

Pay: \$12/hr

Hours: 6

Distance: 3km

Description: Assist with crop harvesting.

Reply:

1. Apply
2. Back
3. Cancel

Set state:

python



Copy code

```
self.store.update_state(phone, "job_details_view")
self.store.update_user(phone, {"selected_job": "job_001"})
```

● If user replies "1 → Apply"

Save into JSON:

python

 Copy code

```
self.store.add_application(phone, job_id)
self.store.update_state(phone, "job_action")
```

Reply:

pgsql

 Copy

Your application **for** Harvest Worker has been submitted! 🎉
We will **notify** the farm **owner**.

💡 APPLICATION WRITING LOGIC

python

 Copy

```
def add_application(self, phone, job_id):
    data = self._load()

    application = {
        "user": phone,
        "job_id": job_id,
        "status": "applied",
        "timestamp": datetime.now().isoformat()
    }

    data.setdefault("applications", []).append(application)
    self._save(data)
```

🎯 WHAT YOU HAVE NOW

- ✓ Full user registration flow
- ✓ State machine architecture

- ✓ JSON datastore CRUD
 - ✓ Job recommendation algorithm
 - ✓ WhatsApp interaction flow
 - ✓ Job application saving system
-



NEXT STEP (OPTIONAL)

If you want, I can generate:

◆ The FULL ready-to-run chatbot project with

- Flask
- Twilio WhatsApp webhook