**STEP 1 :**

!pip install --quiet streamlit langchain langchain-google-genai google-generativeai python-dotenv pyngrok googlemaps geopy pandas pypdf faiss-cpu python-docx

!pip install pandas

!pip install pdfplumber

!pip install geopy

!pip install langchain-community

**STEP 2 :**

from langchain\_community.vectorstores import FAISS

import textwrap, os

GEMINI\_KEY = getpass("Enter your GOOGLE\_API\_KEY (Gemini) (press Enter to skip if you will set later): ")

MAPS\_KEY  = getpass("Enter your MAPS\_API\_KEY (Google Maps) [press Enter to skip]: ")

EMAIL\_SENDER = getpass("Enter EMAIL\_SENDER (e.g. yourapp@gmail.com) [press Enter to skip]: ")

EMAIL\_PASS = getpass("Enter EMAIL\_PASS (app password) [press Enter to skip]: ")

env\_text = textwrap.dedent(f"""

GOOGLE\_API\_KEY={GEMINI\_KEY}

MAPS\_API\_KEY={MAPS\_KEY}

EMAIL\_SENDER={EMAIL\_SENDER}

EMAIL\_PASS={EMAIL\_PASS}

""").strip()

with open(".env", "w") as f:

    f.write(env\_text)

print("✅ .env written (keys NOT shown). If you skipped any, you can edit .env later.")

**STEP 3 :**

%%writefile agent.py

# ------------------------------ agent.py ------------------------------

# FindMyStore core + RAG (PDF/DOCX/TXT indexing) + LangChain tools

# + Persistent Alerts (SQLite) + Email notifications (SMTP)

import os, json, random, sqlite3, smtplib

from typing import List, Dict, Any, Tuple

from email.mime.text import MIMEText

from dotenv import load\_dotenv

load\_dotenv()

GOOGLE\_API\_KEY = os.getenv("GOOGLE\_API\_KEY", "")

MAPS\_API\_KEY   = os.getenv("MAPS\_API\_KEY", "")

EMAIL\_SENDER   = os.getenv("EMAIL\_SENDER", "")

EMAIL\_PASS     = os.getenv("EMAIL\_PASS", "")

# LangChain + Gemini LLM

from langchain\_google\_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings

from langchain.agents import initialize\_agent, Tool, AgentType

# Utilities for geo + data

import pandas as pd

from geopy.distance import geodesic

# Google Maps client (if key provided)

try:

    import googlemaps

    gmaps = googlemaps.Client(key=MAPS\_API\_KEY) if MAPS\_API\_KEY else None

except Exception:

    gmaps = None

# --------- Category mapping ----------

CATEGORY\_TO\_TYPE = {

    "grocery": "grocery\_or\_supermarket",

    "pharmacy": "pharmacy",

    "electronics": "electronics\_store",

    "clothing": "clothing\_store",

    "bakery": "bakery",

    "restaurant": "restaurant",

}

# ---------------- Fallback store data ----------------

stores\_fallback = [

    {"id": 1, "name": "SmartMart Jubilee Hills", "city": "Hyderabad", "category": "grocery",

     "hours": "9am-9pm", "lat": 17.433, "lng": 78.403, "rating": 4.5, "verified": True},

    {"id": 2, "name": "MediCare Pharmacy Banjara", "city": "Hyderabad", "category": "pharmacy",

     "hours": "24/7", "lat": 17.412, "lng": 78.448, "rating": 4.3, "verified": True},

    {"id": 3, "name": "ElectroHub Secunderabad", "city": "Hyderabad", "category": "electronics",

     "hours": "10am-8pm", "lat": 17.444, "lng": 78.501, "rating": 4.1, "verified": False},

    {"id": 4, "name": "StyleStreet Hitech City", "city": "Hyderabad", "category": "clothing",

     "hours": "11am-9pm", "lat": 17.452, "lng": 78.381, "rating": 4.6, "verified": True},

]

# ---------------- Inventory ----------------

inventory: Dict[int, Dict[str, Dict[str, float]]] = {

    1: {"XYZ Shampoo": {"qty": 12, "price": 150}, "Milk Lotion": {"qty": 0, "price": 199}, "Rice 10kg": {"qty": 8, "price": 489}},

    2: {"XYZ Shampoo": {"qty": 4, "price": 155}, "Milk Lotion": {"qty": 7, "price": 189}},

    3: {"Laptop Bag": {"qty": 6, "price": 899}, "USB Cable": {"qty": 15, "price": 149}},

    4: {"T-Shirt": {"qty": 10, "price": 399}},

}

dynamic\_inventory: Dict[int, Dict[str, Dict[str, float]]] = {}

\_last\_store\_cache: List[Dict[str, Any]] = []

# ---------------- Helpers: geo + links ----------------

def geocode\_city(city: str) -> Tuple[float, float]:

    if gmaps:

        try:

            ge = gmaps.geocode(city)

            if ge and ge[0]["geometry"]["location"]:

                loc = ge[0]["geometry"]["location"]

                return loc["lat"], loc["lng"]

        except Exception:

            pass

    return (17.3850, 78.4867)  # Hyderabad fallback

def maps\_place\_link(lat: float, lng: float, place\_id: str = None) -> str:

    if place\_id:

        return f"https://www.google.com/maps/search/?api=1&query=Google&query\_place\_id={place\_id}"

    return f"https://www.google.com/maps?q={lat},{lng}"

def maps\_directions\_link(dest\_lat: float, dest\_lng: float, place\_id: str = None, origin: str = None) -> str:

    base = "https://www.google.com/maps/dir/?api=1"

    parts = [base, f"destination={dest\_lat},{dest\_lng}"]

    if place\_id:

        parts.append(f"destination\_place\_id={place\_id}")

    if origin:

        parts.append(f"origin={origin}")

    parts.append("travelmode=driving")

    return "&".join(parts)

# ---------------- Fetch stores ----------------

def fetch\_stores(city: str, category: str = None, radius\_km: float = 6.0, open\_now: bool = False) -> List[Dict[str, Any]]:

    global \_last\_store\_cache

    results: List[Dict[str, Any]] = []

    if gmaps:

        lat, lng = geocode\_city(city)

        ptype = CATEGORY\_TO\_TYPE.get(category or "", "store")

        try:

            places = gmaps.places\_nearby(

                location=(lat, lng),

                radius=int(radius\_km \* 1000),

                type=ptype if ptype != "store" else None,

                open\_now=open\_now if ptype != "store" else None,

            )

            pid = 1

            for p in places.get("results", []):

                loc = p.get("geometry", {}).get("location", {})

                rating = float(p.get("rating", 4.2))

                urt = int(p.get("user\_ratings\_total", 20))

                results.append({

                    "id": pid,

                    "place\_id": p.get("place\_id"),

                    "name": p.get("name"),

                    "city": city,

                    "category": category or "general",

                    "hours": "—",

                    "lat": loc.get("lat"),

                    "lng": loc.get("lng"),

                    "address": p.get("vicinity", ""),

                    "rating": rating,

                    "verified": True if (rating >= 4.3 and urt >= 30) else False,

                })

                pid += 1

        except Exception:

            results = [s for s in stores\_fallback if s["city"].lower() == city.lower()]

            if category:

                results = [s for s in results if s["category"] == category]

    else:

        results = [s for s in stores\_fallback if s["city"].lower() == city.lower()]

        if category:

            results = [s for s in results if s["category"] == category]

    \_last\_store\_cache = results[:]

    return results

# ---------------- Inventory helpers ----------------

def ensure\_dynamic\_inventory(store\_id: int, product: str):

    di = dynamic\_inventory.setdefault(store\_id, {})

    if product not in di:

        di[product] = {"qty": random.randint(0, 12),

                      "price": random.choice([79, 99, 129, 149, 199, 249, 299, 399, 899, 999])}

def get\_inventory(store\_id: int) -> Dict[str, Dict[str, float]]:

    if store\_id in inventory:

        return inventory[store\_id]

    return dynamic\_inventory.setdefault(store\_id, {})

def check\_stock(store\_id: int, product: str) -> Dict[str, Any]:

    inv = get\_inventory(store\_id)

    if product not in inv:

        ensure\_dynamic\_inventory(store\_id, product)

        inv = get\_inventory(store\_id)

    data = inv.get(product, {"qty": 0, "price": None})

    return {"store\_id": store\_id, "product": product, "qty": int(data["qty"]), "price": data["price"]}

def find\_cheapest(product: str, max\_price: float = None) -> Dict[str, Any]:

    search\_space = \_last\_store\_cache[:] or stores\_fallback[:]

    best = None

    for s in search\_space:

        info = check\_stock(s["id"], product)

        if info["qty"] > 0 and info["price"] is not None:

            if max\_price is not None and info["price"] > max\_price:

                continue

            if best is None or info["price"] < best["price"]:

                best = {\*\*info, "store": s}

    return best or {"message": f"No available '{product}' found within criteria."}

def shopping\_list\_optimize(products: List[str]) -> Dict[str, Any]:

    needed = set([p.strip() for p in products if p.strip()])

    if not needed:

        return {"message": "No items in shopping list."}

    search\_space = \_last\_store\_cache[:] or stores\_fallback[:]

    offerings = {}

    for s in search\_space:

        sid = s["id"]

        o = {}

        for item in needed:

            info = check\_stock(sid, item)

            if info["qty"] > 0 and info["price"] is not None:

                o[item] = info["price"]

        if o:

            offerings[sid] = o

    chosen = []

    remaining = set(needed)

    total\_cost = 0.0

    while remaining:

        best\_sid, best\_cover, best\_score = None, set(), float("inf")

        for sid, offer in offerings.items():

            cover = remaining.intersection(offer.keys())

            if cover:

                cost = sum(offer[i] for i in cover)

                score = cost / len(cover)

                if score < best\_score:

                    best\_sid, best\_cover, best\_score = sid, cover, score

        if not best\_sid:

            break

        chosen.append({

            "store\_id": best\_sid,

            "items": {i: offerings[best\_sid][i] for i in best\_cover},

            "subtotal": sum(offerings[best\_sid][i] for i in best\_cover),

        })

        total\_cost += chosen[-1]["subtotal"]

        remaining -= best\_cover

        offerings.pop(best\_sid, None)

    return {

        "covered\_all": len(remaining) == 0,

        "plan": chosen,

        "not\_found": list(remaining),

        "total\_cost": round(total\_cost, 2),

    }

# ---------------- Alerts (persistent SQLite + email) ----------------

DB\_PATH = "alerts.db"

def init\_alerts\_db():

    conn = sqlite3.connect(DB\_PATH, check\_same\_thread=False)

    c = conn.cursor()

    c.execute("""

        CREATE TABLE IF NOT EXISTS alerts (

            id INTEGER PRIMARY KEY AUTOINCREMENT,

            product TEXT NOT NULL,

            city TEXT,

            email TEXT

        )

    """)

    conn.commit()

    return conn

\_alerts\_conn = init\_alerts\_db()

def subscribe\_alert(product: str, city: str, email: str = None) -> Dict[str, Any]:

    c = \_alerts\_conn.cursor()

    c.execute(

        "INSERT INTO alerts (product, city, email) VALUES (?, ?, ?)",

        (product.strip(), (city or "").strip(), (email or "").strip() or None)

    )

    \_alerts\_conn.commit()

    return {"message": f"Subscribed to '{product}' alerts in {city} ({'with email' if email else 'no email'})"}

def get\_all\_subscriptions() -> List[Dict[str, Any]]:

    c = \_alerts\_conn.cursor()

    c.execute("SELECT id, product, city, email FROM alerts ORDER BY id DESC")

    rows = c.fetchall()

    return [{"id": r[0], "product": r[1], "city": r[2], "email": r[3]} for r in rows]

def unsubscribe\_alert(alert\_id: int) -> Dict[str, Any]:

    c = \_alerts\_conn.cursor()

    c.execute("DELETE FROM alerts WHERE id = ?", (alert\_id,))

    \_alerts\_conn.commit()

    return {"message": f"Unsubscribed alert id {alert\_id}."}

from email.mime.multipart import MIMEMultipart

from email.mime.text import MIMEText

def send\_email\_alert(to\_email: str, product: str, city: str) -> bool:

    if not EMAIL\_SENDER or not EMAIL\_PASS:

        print("Email sender or pass not configured; cannot send email.")

        return False

    try:

        msg = MIMEMultipart()

        msg['From'] = EMAIL\_SENDER

        msg['To'] = to\_email

        msg['Subject'] = f"FindMyStore — '{product}' Restock Alert"

        body = f"Good news! '{product}' is back in stock in {city}.\n\n— FindMyStore"

        msg.attach(MIMEText(body, 'plain'))

        with smtplib.SMTP\_SSL('smtp.gmail.com', 465) as server:

            server.login(EMAIL\_SENDER, EMAIL\_PASS)

            server.send\_message(msg)

        return True

    except Exception as e:

        print("Error sending email:", e)

        return False

def simulate\_restock(product: str, store\_id: int, qty: int = 10) -> Dict[str, Any]:

    inv = get\_inventory(store\_id)

    if product not in inv:

        inv[product] = {"qty": qty, "price": random.choice([99, 129, 149, 199, 249, 299])}

    else:

        inv[product]["qty"] = max(0, inv[product]["qty"]) + qty

    c = \_alerts\_conn.cursor()

    c.execute("SELECT id, product, city, email FROM alerts WHERE lower(product)=?", (product.lower(),))

    rows = c.fetchall()

    notified = 0

    for r in rows:

        \_id, p, city, email = r

        if email:

            ok = send\_email\_alert(email, product, city or "your city")

            if ok:

                notified += 1

    return {"message": f"Restocked '{product}' at store {store\_id}. Alerts notified (emails sent): {notified}."}

# ---------------- UI Wrappers ----------------

def ui\_search\_stores(city: str, category: str = None, radius\_km: float = 6.0, open\_now: bool = False):

    return fetch\_stores(city, category, radius\_km, open\_now)

def ui\_compare\_prices(product: str, city: str = None, category: str = None, radius\_km: float = 6.0):

    if city:

        fetch\_stores(city, category, radius\_km)

    rows = []

    for s in \_last\_store\_cache or stores\_fallback:

        info = check\_stock(s["id"], product)

        rows.append({

            "store\_id": s["id"],

            "store": s.get("name"),

            "city": s.get("city"),

            "category": s.get("category"),

            "verified": s.get("verified", False),

            "rating": s.get("rating"),

            "qty": info["qty"],

            "price": info["price"],

            "map": maps\_place\_link(s.get("lat"), s.get("lng"), s.get("place\_id")),

        })

    df = pd.DataFrame(rows)

    if not df.empty:

        df = df.sort\_values(by=["qty","price"], ascending=[False, True])

    return df

def ui\_shopping\_list(products\_csv: str, city: str = None, category: str = None, radius\_km: float = 6.0):

    if city:

        fetch\_stores(city, category, radius\_km)

    items = [x.strip() for x in products\_csv.split(",") if x.strip()]

    return shopping\_list\_optimize(items)

def ui\_get\_directions(store\_id: int, origin: str = None) -> str:

    s = next((x for x in (\_last\_store\_cache or stores\_fallback) if x["id"] == store\_id), None)

    if not s:

        return "Store not found."

    return maps\_directions\_link(s["lat"], s["lng"], s.get("place\_id"), origin)

# ---------------- RAG ----------------

from pypdf import PdfReader

import docx

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.vectorstores import FAISS

rag\_store = None

def load\_document(file\_path: str) -> str:

    text = ""

    lower = file\_path.lower()

    try:

        if lower.endswith(".pdf"):

            reader = PdfReader(file\_path)

            for page in reader.pages:

                t = page.extract\_text()

                if t:

                    text += t + "\n"

        elif lower.endswith(".docx"):

            doc = docx.Document(file\_path)

            for p in doc.paragraphs:

                if p.text:

                    text += p.text + "\n"

        elif lower.endswith(".txt"):

            with open(file\_path, "r", encoding="utf-8") as f:

                text = f.read()

    except Exception:

        return ""

    return text

def build\_rag\_index(file\_path: str):

    global rag\_store

    raw\_text = load\_document(file\_path)

    if not raw\_text.strip():

        return {"message": "No text extracted from document."}

    splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)

    chunks = splitter.split\_text(raw\_text)

    embeddings = GoogleGenerativeAIEmbeddings(model="models/embedding-001", google\_api\_key=GOOGLE\_API\_KEY)

    rag\_store = FAISS.from\_texts(chunks, embeddings)

    return {"message": f"Indexed {len(chunks)} chunks from document."}

def query\_rag(question: str, k: int = 3) -> str:

    global rag\_store

    if rag\_store is None:

        return "No document uploaded yet. Please upload a PDF/DOCX/TXT first."

    docs = rag\_store.similarity\_search(question, k=k)

    if not docs:

        return "No relevant content found in the uploaded document."

    context = "\n\n".join([d.page\_content for d in docs])

    prompt = f"""You are a helpful assistant.

Use only the following document context to answer the question.

If the answer is not in the context, say you don't know.

Context:

{context}

Question: {question}

Answer:"""

    try:

        response = llm.invoke(prompt)

        if isinstance(response, dict):

            return response.get("output", response.get("output\_text", str(response)))

        if hasattr(response, "content"):

            return response.content

        return str(response)

    except Exception as e:

        return f"Error generating answer: {e}"

# ---------------- LLM + Tools ----------------

llm = ChatGoogleGenerativeAI(

    model="models/gemini-1.5-pro-latest",

    temperature=0,

    max\_output\_tokens=1024,

    google\_api\_key=GOOGLE\_API\_KEY,

)

tools = [

    Tool(name="search\_stores", func=ui\_search\_stores, description="Search stores by city/category."),

    Tool(name="compare\_prices", func=ui\_compare\_prices, description="Compare prices for a product."),

    Tool(name="shopping\_list", func=ui\_shopping\_list, description="Optimize shopping list across stores."),

    Tool(name="get\_directions", func=ui\_get\_directions, description="Get Google Maps directions to a store."),

]

agent = initialize\_agent(tools, llm, agent\_type=AgentType.OPENAI\_FUNCTIONS, verbose=True)

def run\_agent(query: str) -> str:

    return agent.run(query)

**STEP 4 :**

   %%writefile app.py

# ------------------------------ app.py ------------------------------

# Streamlit front-end for FindMyStore + RAG + persistent alerts + email

import streamlit as st

import pandas as pd

# Import UI functions and chatbot from agent.py

from agent import (

    ui\_search\_stores, ui\_compare\_prices, ui\_shopping\_list,

    ui\_get\_directions, run\_agent, subscribe\_alert, simulate\_restock,

    build\_rag\_index, query\_rag, get\_all\_subscriptions, unsubscribe\_alert

)

# Basic page settings

st.set\_page\_config(page\_title="FindMyStore", layout="wide")

st.title("🏬 FindMyStore — Demo (with RAG & Alerts)")

# Sidebar filters — applied to multiple tabs

with st.sidebar:

    st.header("Search Filters")

    city = st.text\_input("City", value="Hyderabad")

    category = st.selectbox("Category", ["", "grocery", "pharmacy", "electronics", "clothing", "bakery", "restaurant"])

    radius = st.slider("Radius (km)", 1, 20, 6)

    open\_now = st.checkbox("Open now", value=False)

    st.caption("Tip: Leave category blank for a broad search.")

# Tabs for features

tabs = st.tabs(["🗺️ Nearby Stores", "🔍 Product Compare", "🛒 Shopping List", "🔔 Alerts", "🤖 Chatbot (with RAG)"])

# -------------------- Tab 1: Nearby Stores --------------------

with tabs[0]:

    st.subheader("Nearby Store Finder + Maps Integration")

    if st.button("Find Stores", type="primary"):

        results = ui\_search\_stores(city, category or None, radius, open\_now)

        if not results:

            st.warning("No stores found. Try a different category or radius.")

        else:

            st.success(f"Found {len(results)} store(s).")

            df\_map = pd.DataFrame([{"lat": r["lat"], "lon": r["lng"]} for r in results if r.get("lat") and r.get("lng")])

            if not df\_map.empty:

                st.map(df\_map, size=100)

            for s in results:

                with st.container():

                    left, right = st.columns([0.7, 0.3])

                    with left:

                        badge = "✅ Verified" if s.get("verified") else "⚪ Unverified"

                        stars = "⭐" \* int(round(s.get("rating", 4.0)))

                        st.markdown(f"\*\*{s['name']}\*\*  \n{badge} · {stars}")

                        st.write(f"Category: {s.get('category','—')}  |  City: {s.get('city','—')}")

                        if s.get("address"):

                            st.write(f"Address: {s['address']}")

                        if s.get("hours"):

                            st.write(f"Hours: {s['hours']}")

                    with right:

                        link = f"https://www.google.com/maps?q={s.get('lat')},{s.get('lng')}"

                        st.markdown(f"[📍 View on Map]({link})")

# -------------------- Tab 2: Product Compare --------------------

with tabs[1]:

    st.subheader("Product Availability + Price Comparison")

    col1, col2 = st.columns([0.6,0.4])

    with col1:

        product = st.text\_input("Product name", value="XYZ Shampoo")

        if st.button("Compare Prices"):

            df = ui\_compare\_prices(product, city, category or None, radius)

            if df.empty:

                st.warning("No data.")

            else:

                cheapest = df[df["qty"] > 0].sort\_values(by="price", ascending=True).head(1)

                if not cheapest.empty:

                    c = cheapest.iloc[0]

                    st.success(f"Cheapest: \*\*{c['store']}\*\* — ₹{int(c['price'])} (qty {int(c['qty'])})")

                st.dataframe(df, use\_container\_width=True)

    with col2:

        st.info("Tip: pick a Category in the sidebar to narrow the store set used for comparison.")

# -------------------- Tab 3: Shopping List --------------------

with tabs[2]:

    st.subheader("Smart Shopping List + Budget Mode")

    items = st.text\_area("Enter items (comma-separated)", value="Rice 10kg, XYZ Shampoo, Milk Lotion")

    if st.button("Optimize Shopping List"):

        res = ui\_shopping\_list(items, city, category or None, radius)

        if "message" in res:

            st.warning(res["message"])

        else:

            if res["plan"]:

                for p in res["plan"]:

                    with st.container():

                        st.markdown(f"\*\*Store #{p['store\_id']}\*\* — Subtotal: ₹{int(p['subtotal'])}")

                        lines = [f"- {k}: ₹{int(v)}" for k, v in p["items"].items()]

                        st.markdown("\n".join(lines))

                st.success(f"Total estimated cost: ₹{int(res['total\_cost'])}")

            if res["not\_found"]:

                st.error(f"Not found: {', '.join(res['not\_found'])}")

            if res["covered\_all"]:

                st.caption("✅ All items covered")

# -------------------- Tab 4: Alerts (persistent + email) --------------------

with tabs[3]:

    st.subheader("Real-Time Stock & Deal Alerts")

    st.markdown("Subscribe to alerts (persistent). Provide an email to get email notifications when a product is restocked.")

    col1, col2 = st.columns([0.6, 0.4])

    with col1:

        sub\_prod = st.text\_input("Product to subscribe", value="XYZ Shampoo")

        sub\_city = st.text\_input("City for alerts", value=city)

        sub\_email = st.text\_input("Your email (optional, for notifications)", value="")

        if st.button("Subscribe"):

            res = subscribe\_alert(sub\_prod, sub\_city, sub\_email.strip() or None)

            st.success(res["message"])

    with col2:

        sim\_prod = st.text\_input("Simulate restock product", value="XYZ Shampoo")

        sim\_store = st.number\_input("Store ID to restock", value=1, min\_value=1, step=1)

        if st.button("Simulate Restock"):

            res = simulate\_restock(sim\_prod, int(sim\_store))

            st.success(res["message"])

    st.markdown("### Active subscriptions")

    subs = get\_all\_subscriptions()

    if subs:

        df\_subs = pd.DataFrame(subs)

        st.dataframe(df\_subs, use\_container\_width=True)

        to\_unsub = st.number\_input("Unsubscribe alert id (enter id from table)", value=0, min\_value=0, step=1)

        if to\_unsub:

            st.success(unsubscribe\_alert(int(to\_unsub))["message"])

    else:

        st.info("No subscriptions yet.")

    st.caption("Notes: Emails require EMAIL\_SENDER and EMAIL\_PASS set in .env. Use app passwords for Gmail.")

# -------------------- Tab 5: Chatbot (with RAG) --------------------

with tabs[4]:

    st.subheader("Voice + AI Chatbot Assistant (with RAG document QA)")

    st.caption("You can upload a PDF/DOCX/TXT and then ask questions about it. The chatbot still has store tools too.")

    uploaded\_file = st.file\_uploader("Upload PDF, DOCX, or TXT (optional)", type=["pdf","docx","txt"])

    if uploaded\_file:

        save\_path = f"uploaded\_{uploaded\_file.name}"

        with open(save\_path, "wb") as f:

            f.write(uploaded\_file.read())

        st.info("Indexing uploaded document (this may take a few seconds)...")

        msg = build\_rag\_index(save\_path)

        st.success(msg["message"])

        st.session\_state["uploaded\_file\_path"] = save\_path

    use\_doc\_only = st.checkbox("Answer from uploaded document only (skip store tools)", value=False)

    query = st.text\_input("Your message")

    if st.button("Ask 🤖"):

        if not query.strip():

            st.warning("Type something first 🙂")

        else:

            if use\_doc\_only:

                answer = query\_rag(query)

                st.markdown("\*\*Answer (document-only):\*\*")

                st.write(answer)

            else:

                with st.spinner("Thinking..."):

                    answer = run\_agent(query)

                st.markdown("\*\*Assistant:\*\*")

                st.write(answer)

# ------------------------------ end app.py ------------------------------

**STEP 5 :**

# STEP 5: Start Streamlit (and optional ngrok)

from getpass import getpass

import subprocess, time

from pyngrok import ngrok

NGROK\_TOKEN = getpass("Paste your ngrok authtoken (or press Enter to skip): ")

if NGROK\_TOKEN:

    # configure ngrok if provided

    !ngrok config add-authtoken {NGROK\_TOKEN}

# Kill previous Streamlit processes (ignore errors)

!pkill -f "streamlit run app.py" 2>/dev/null || true

# Start Streamlit app

proc = subprocess.Popen(["streamlit", "run", "app.py", "--server.port", "8501"])

# Wait a moment and create tunnel

time.sleep(3)

public\_url = ngrok.connect(8501)

print("🌐 Public App URL:", public\_url)

print("If the URL shows a 502 initially, wait ~5s and refresh.")