1 — App setup & imports

```
import os, json, time

from flask import Flask, render_template, request, redirect, url_for, flash, jsonify, send_from_directory

from flask_login import LoginManager, login_user, logout_user, login_required, current_user

from werkzeug.security import check_password_hash, generate_password_hash

from werkzeug.utils import secure_filename

from database import init_db, db, User, ChatHistory

from chatbot_model import process_message, load_kb, KB_PATH

from utils.safety import contains_blocked, sanitize_output

from PIL import Image

import io
```

- Imports core Flask modules, auth helpers (flask_login), password hashing, secure filename helper.
- database module is expected to provide DB setup functions and ORM models (User, ChatHistory, db).
- chatbot_model.process_message is your app's logic for generating the bot reply.
- utils.safety contains checks for blocked content and a sanitizer for output.
- PIL.Image + io are used to read/process uploaded images.

```
UPLOAD_FOLDER = os.path.join(os.path.dirname(__file__), "uploads")

os.makedirs(UPLOAD_FOLDER, exist_ok=True)
```

- Creates uploads/ directory next to app.py if it doesn’t exist — used to save uploaded images.

app = Flask(__name__, static_folder="static", template_folder="templates")

app.config["SECRET_KEY"] = os.getenv("FLASK_SECRET_KEY", "super_secret_key")

app.config["UPLOAD_FOLDER"] = UPLOAD_FOLDER

- Creates the Flask app and sets secret key (used by sessions/flash). In production you should set FLASK_SECRET_KEY env var.

init_db(app)

- Initializes DB — likely registers models, creates tables and binds app context.

2 — Login manager

login_manager = LoginManager()

login_manager.init_app(app)

login_manager.login_view = "login"

- Sets up flask_login to manage sessions. login_view used to redirect unauthorized users.

@login_manager.user_loader

def load_user(user_id):

  return db.session.get(User, int(user_id))

- Tells flask_login how to retrieve a User object from stored session user id.

3 — Index route

@app.route("/")

```
def index():
    recent_users = []
    if current_user.is_authenticated and current_user.role == "admin":
        recent_users = User.query.order_by(User.id.desc()).limit(20).all()
    return render_template("index.html", recent_users=recent_users)
```

- Serves the home page. If an admin is logged in, shows last 20 users (for admin convenience).

4 — Register

```
@app.route("/register", methods=["GET", "POST"])
def register():
    if request.method == "POST":
        email = request.form["email"].strip().lower()
        if User.query.filter_by(email=email).first():
            flash("Email already registered", "warning");
            return redirect(url_for("register"))
        user = User(
            email=email,

password=generate_password_hash(request.form["password"]),
            name=request.form.get("name", ""),
            primary_crop=request.form.get("primary_crop", ""),
            region=request.form.get("region", ""),
            preferred_language=request.form.get("preferred_language",
"en")
        )
```

```
        db.session.add(user);
        db.session.commit()
        flash("Registration successful — please log in", "success")
        return redirect(url_for("login"))
    return render_template("register.html")
```

- GET: show registration form.
- POST: validate uniqueness, create User record, hash password with generate_password_hash, commit to DB, redirect to login.
- Uses flash() to show feedback messages in templates.

5 — Login & Logout

```
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        email = request.form["email"].strip().lower()
        user = User.query.filter_by(email=email).first()
        if user and check_password_hash(user.password,
request.form["password"]):
            login_user(user)
            flash("Welcome back, " + (user.name or "Farmer") + "!",
"success")
            return redirect(url_for("index"))
        flash("Invalid credentials", "danger")
    return render_template("login.html")
```

- Authenticates user by comparing hashed password.

- On success calls login_user(user) which persists login in session.
- Logout:

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    flash("Logged out", "info")
    return redirect(url_for("index"))
```

- @login_required ensures only logged-in users can call logout (minor — not strictly necessary).

6 — Profile editing

```
@app.route("/profile", methods=["GET", "POST"])
@login_required
def profile():
    if request.method == "POST":
        current_user.name = request.form.get("name", "")
        current_user.primary_crop = request.form.get("primary_crop", "")
        current_user.region = request.form.get("region", "")
        current_user.preferred_language = request.form.get("preferred_language", "en")
        db.session.commit()
        flash("Profile updated", "success")
    return render_template("profile.html")
```

- Lets authenticated users update profile fields (changes are committed directly on current_user model).

7 — Chat API endpoint

```
@app.route("/api/chat", methods=["POST"])
def api_chat():
    try:
        data = request.get_json() or {}
        message = (data.get("message") or "").strip()
        if not message:
            return jsonify({"response": "Please type a question."})
        if contains_blocked(message):
            return jsonify({"response": "Sorry, message contains prohibited content."}), 400


        user_profile = {
            "id": current_user.id if current_user.is_authenticated else None,
            "primary_crop": current_user.primary_crop if current_user.is_authenticated else None,
            "region": current_user.region if current_user.is_authenticated else None,
            "preferred_language": current_user.preferred_language if current_user.is_authenticated else None
        }
        reply = process_message(user_profile, message)
```

```
        reply = sanitize_output(reply)
        ch = ChatHistory(user_id=user_profile["id"],
user_message=message, bot_response=reply)
        db.session.add(ch);
        db.session.commit()
        return jsonify({"response": reply})
    except Exception as e:
        print("Error /api/chat:", e)
        return jsonify({"response": "Internal server error"}), 500
```

- Receives JSON {message: "..."}.
- Validates non-empty and checks for blocked content via contains_blocked.
- Builds user_profile from current_user if authenticated (used to personalize bot reply).
- Calls process_message(user_profile, message) — your chatbot core logic. Then sanitizes the reply for safety.
- Stores the interaction in ChatHistory (user_id optional if anonymous).
- Returns JSON with the bot reply.
- On error returns 500 and prints the exception (consider logging instead of print in prod).

8 — Admin routes

- @app.route("/admin"): only accessible to admin role; shows list of users, recent chats and reads the KB file from KB_PATH. Uses render_template("admin_dashboard.html", ...).

- @app.route("/admin/user/<int:user_id>"): view individual user and their chat history.
- @app.route("/admin/edit_kb", methods=["POST"]): allows admin to paste JSON content for the KB and writes to KB_PATH. Validates JSON with json.loads() and writes with ensure_ascii=False.
- @app.route("/admin/upload_kb_csv", methods=["POST"]): CSV uploader:
    - Saves uploaded CSV to uploads/.
    - Uses csv.DictReader to parse rows with expected columns keywords,answer_en,answer_hi,answer_ta.
    - Normalizes keywords into list (split by comma).
    - Loads existing KB JSON (if any), appends new rows, writes back.
    - Flashes count or error.
- @app.route("/admin/delete_user/<int:user_id>", methods=["POST"]):
    - Admin-only deletion; protects against deleting another admin. Deletes user record.
- @app.route("/admin/clear_chats", methods=["POST"]):
    - Admin-only; deletes all ChatHistory records, commits, flashes success.

Important: Admin routes all check current_user.role != "admin" and redirect/return unauthorized — good.

9 — Image analysis config

ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif', 'webp'}

MAX_FILE_SIZE = 5 * 1024 * 1024 # 5MB

- Whitelists file extensions and sets file size limit.

```
def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

- Simple extension-based check.

10 — /api/analyze-image endpoint (core image logic)

```
@app.route("/api/analyze-image", methods=["POST"])
@login_required
def analyze_image():
    try:
        if 'image' not in request.files:
            return jsonify({"error": "No image file provided"}), 400
        file = request.files['image']
        if file.filename == '':
            return jsonify({"error": "No file selected"}), 400
        if not file or not allowed_file(file.filename):
            return jsonify({"error": "Invalid file type. Allowed: PNG, JPG, JPEG, GIF, WebP"}), 400

        # Check file size
        file.seek(0, 2)
        file_size = file.tell()
        file.seek(0)
        if file_size > MAX_FILE_SIZE:
```

```
            return jsonify({"error": "File too large. Maximum size is
5MB"}), 400

        text_message = request.form.get('message', '').strip()
        image_data = file.read()
        img = Image.open(io.BytesIO(image_data))

        image_info = {
            'filename': secure_filename(file.filename),
            'size': file_size,
            'dimensions': img.size,
            'format': img.format,
            'mode': img.mode
        }

        analysis_result = analyze_image_content(img, text_message)

        user_message = f"[Image: {image_info['filename']}]
{text_message}" if text_message else f"[Image:
{image_info['filename']}]"
        if current_user.is_authenticated:
            ch = ChatHistory(user_id=current_user.id,
user_message=user_message,
bot_response=analysis_result['response'])
            db.session.add(ch)
```

```
            db.session.commit()


        return jsonify({
            "success": True,
            "response": analysis_result['response'],
            "analysis": analysis_result['analysis'],
            "image_info": image_info
        })
    except Exception as e:
        print("Image analysis error:", e)
        return jsonify({"error": "Image analysis failed", "message":
str(e)}), 500
```

- Requires login (@login_required) — only authenticated users can analyze images.
- Validates presence, filename, allowed type, and size by seeking to end to compute bytes length.
- Reads raw bytes and opens the image via PIL for processing (no file saved to disk in this code).
- Builds image_info metadata: filename, file size, dimensions, format, mode.
- Calls analyze_image_content(img, text_message) to get analysis and response text.
- Saves the chat history entry with ChatHistory.
- Returns JSON including response (user-facing text), analysis (numeric/structured results), and image_info.

11 — Image analysis functions

analyze_image_content(img, user_question=""):

- Converts image to RGB and iterates over pixels to compute basic color statistics.
    - green_pixels: counts pixels where green component dominates (g > r + 20 and g > b + 20).
    - brown_pixels, yellow_pixels use heuristics for brown/yellow detection.
- Computes ratios and sets health_status:
    - green_ratio > 0.6 → HEALTHY
    - green_ratio > 0.3 → MODERATE
    - else → STRESSED
- Creates analysis dictionary containing percentages and health/confidence.
- Calls generate_image_response(analysis, user_question, img.format) to build a readable response string (with simple markdown-like formatting and agricultural suggestions).
- Returns { "response": ..., "analysis": ... }.

generate_image_response(...):

- Builds an array of response lines containing the summary, color percentages, and actionable suggestions (e.g., check soil moisture, pests).
- Returns a newline-joined string. This is user-facing content returned in JSON.

12 — Static file serving

@app.route("/uploads/<path:filename>")

@login_required

def uploaded_file(filename):

return send_from_directory(app.config['UPLOAD_FOLDER'], filename)

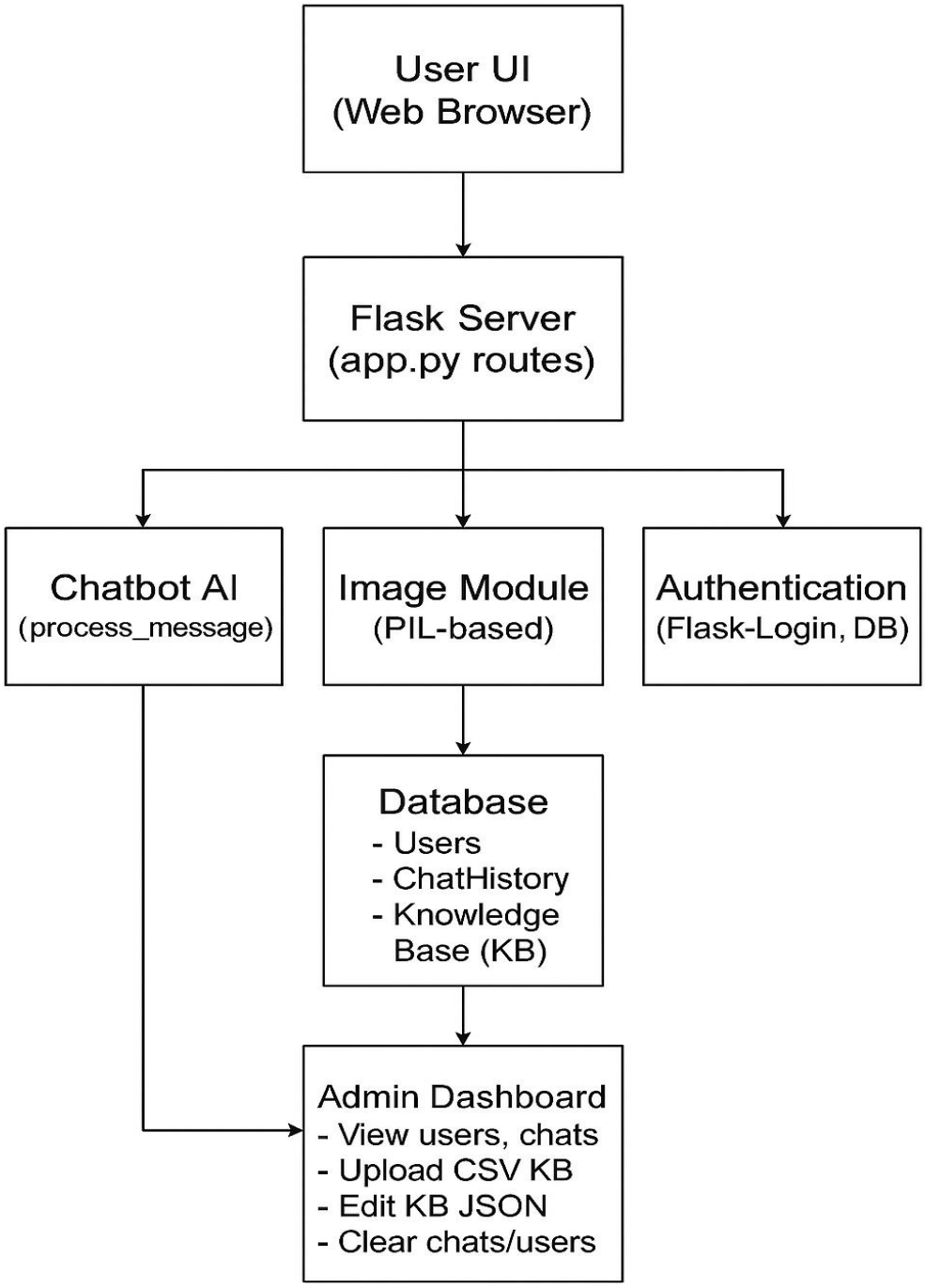
- Serves uploaded files from the uploads/ folder. Route requires login.

13 — App run

if __name__ == "__main__":

app.run(debug=True, host="0.0.0.0")

- Launches Flask dev server listening on all interfaces.


User UI
(Web Browser)
Flask Server
(app.py routes)
Chatbot AI
(process_message)
Image Module
(PIL-based)
Authentication
(Flask-Login, DB)
Database
- Users
- ChatHistory
- Knowledge Base (KB)
Admin Dashboard
- View users, chats
- Upload CSV KB
- Edit KB JSON
- Clear chats/users