**Core Tech Stack**

| **Component** | **Tools & Libraries** | **Why?** |
| --- | --- | --- |
| **Backend** | Python + FastAPI | Lightweight, async-ready, and easy to prototype. Perfect for REST APIs. |
| **Computer Vision** | OpenCV + **pytesseract** (Tesseract OCR wrapper) | OpenCV for image preprocessing (cropping, lighting correction), Tesseract for text extraction. |
| **Language Model** | OpenAI GPT-4/3.5-turbo API (or free alternatives: **Llama 3** via Hugging Face) | GPT-4 for ingredient/calorie analysis. Use Hugging Face if avoiding API costs. |
| **Frontend** | React + Vite (TypeScript recommended) + Axios | Vite for fast setup. Axios for API calls. TypeScript adds safety. |
| **Database** | SQLite (or PostgreSQL for relationships) | SQLite for simplicity (no server setup). Use PostgreSQL if relationships are needed. |
| **API Communication** | FastAPI endpoints + React Query (frontend data fetching) | React Query simplifies caching and error handling. |
| **Containerization** | Docker + Docker Compose | Isolate services (backend, frontend, DB) for easy setup. |

**Key Libraries & Tools**

1. **Backend (Python)**
   * fastapi: API framework.
   * uvicorn: ASGI server.
   * python-multipart: Handle image uploads.
   * pytesseract: Extract text from images.
   * openai or huggingface\_hub: Integrate with GPT or open-source LLMs.
   * pydantic: Data validation for API requests/responses.
   * SQLAlchemy (if using PostgreSQL): ORM for database interactions.
2. **Frontend (React)**
   * react-webcam: Access device camera for scanning.
   * react-router-dom: Basic navigation.
   * zustand or Context API: Simple state management.
   * Material-UI or Chakra UI: Prebuilt components for rapid UI development.
   * react-toastify: Display error/success messages.
3. **Vision Pipeline**
   * **Steps**:
     1. Use OpenCV to preprocess the image (e.g., grayscale, thresholding).
     2. Extract text with pytesseract.
     3. Parse ingredients/calories from OCR text (regex or LLM cleanup).
     4. Send parsed data to GPT for analysis.
4. **Database**
   * For SQLite: Use FastAPI’s built-in support (no ORM needed for simple CRUD).
   * For PostgreSQL: Add asyncpg + SQLAlchemy for async operations.

**Alternatives & Tradeoffs**

* **Python vs. Java**: Stick with Python—faster prototyping, better CV/ML libraries.
* **Tesseract vs. Cloud Vision APIs**: Tesseract is free but less accurate. For a class demo, it’s sufficient. If accuracy is critical, use **Google Cloud Vision API** (free tier available).
* **GPT Alternatives**: Use **Llama 3** (via Hugging Face) to avoid OpenAI costs, but it’ll require more local compute.

**Project Structure**

Copy

/eat-good

├── backend/

│ ├── Dockerfile

│ ├── app/

│ │ ├── main.py (FastAPI routes)

│ │ ├── vision.py (OpenCV + OCR logic)

│ │ ├── models.py (DB models)

│ │ └── gpt\_handler.py (LLM integration)

├── frontend/

│ ├── Dockerfile

│ ├── src/

│ │ ├── components/ (CameraScanner, ResultsDisplay)

│ │ ├── hooks/ (useApiRequest, useCamera)

│ │ └── App.tsx

├── docker-compose.yml

└── README.md

**Docker Setup**

1. **Backend Dockerfile**:

dockerfile

Copy

FROM python:3.11-slim

RUN apt-get update && apt-get install -y tesseract-ocr libgl1

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY ./app /app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]

1. **docker-compose.yml**:

yaml

Copy

version: "3.9"

services:

backend:

build: ./backend

ports:

- "8000:8000"

volumes:

- ./backend/app:/app

frontend:

build: ./frontend

ports:

- "3000:3000"

depends\_on:

- backend

**Workflow for Class Demo**

1. **Scanning**:
   * User points camera at product → React captures image.
   * Frontend sends image to FastAPI /scan endpoint.
2. **Backend Processing**:
   * OpenCV preprocesses the image.
   * Tesseract extracts text (ingredients, calories).
   * GPT evaluates healthiness against user goals (stored in SQLite).
3. **Response**:
   * FastAPI returns JSON like:

json

Copy

{

"product": "Granola Bar",

"calories": 200,

"verdict": "High sugar! Avoid for weight loss.",

"alternatives": ["Almond Butter Bar"]

}

* + React displays results with a friendly UI.

**Tips for Success**

* Start with a **hardcoded example** (skip OCR/GPT initially) to test the frontend-backend flow.
* Use mock GPT responses during development to avoid API costs.
* For OCR, test with clear product labels (e.g., nutrition facts from Google Images).
* Prioritize a **minimal demo** (scan → mock analysis → result) before adding extras.

This stack is **easy to deploy locally**, requires no cloud costs, and focuses on core functionality.