# **MealPlan AI - Codebase Documentation**

This document explains the architecture, configuration, endpoints, data contracts, UI behavior, and operational notes for the MealPlan AI Flask application you shared. It is written to be accurate to the code as provided and includes call examples, gotchas, and suggested fixes for identified issues.

## **1) Overview**

**MealPlan AI** is a small Flask web app that provides:

* Text chat to an OpenAI model, with **Server-Sent Events (SSE)** streaming.
* Optional **text-to-speech (TTS)** of assistant replies via the **Web Speech API**.
* Optional (currently disabled/broken) **structured meal plan generation** endpoint.

The app renders a single page (/) with a chat panel. The frontend streams assistant tokens from /api/chat/stream, shows them in a left/right bubble UI, and (optionally) speaks the reply sentence-by-sentence in real time.

├─ app.py # Flask app & API endpoints

├─ config.py # Application config (env-driven)

├─ templates/

│ ├─ base.html # Base template (Tailwind, frame)

│ └─ index.html # Main page with chat UI

├─ static/

│ ├─ css/custom.css # Custom styles (referenced by base.html)

│ ├─ js/app.js # Frontend logic (chat, SSE, TTS, voice)

│ └─ img/logo.svg # App logo (referenced by base.html)

└─ .env # Environment variables (not committed)

## **3) Requirements**

* **Python** 3.10+ (recommended)
* **Node/npm** NOT required (Tailwind via CDN)
* **OpenAI Python SDK** (modern)  
   pip install openai flask python-dotenv

## **4) Configuration**

### **4.1 Environment variables (.env)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Required** | **Default** | **Description** |
| SECRET\_KEY | No | dev-secret | Flask secret key |
| OPENAI\_API\_KEY | Yes | "" | Your OpenAI API key |
| OPENAI\_MODEL | No | gpt-4o-mini | Default model used server-side |
| MAX\_TOKENS | No | 1200 | Max tokens per completion |
| TEMPERATURE | No | 0.7 | Sampling temperature |

A minimal .env example:

SECRET\_KEY=change-me

OPENAI\_API\_KEY=sk-...

OPENAI\_MODEL=gpt-4o-mini

MAX\_TOKENS=1200

TEMPERATURE=0.7

**4.2 Content Security Policy (CSP)**

Configured in Config.CONTENT\_SECURITY\_POLICY:

* Allows Tailwind via CDN, inline styles, and your own domain.
* connect-src 'self' permits SSE to your own backend origin.
* Adjust if you host assets elsewhere or proxy the API.

**5) Running Locally**

python -m venv .venv

pip install -r requirements.txt # or pip install flask openai python-dotenv

python [app.py](http://app.py)

App listens on <http://0.0.0.0:8080>.

## **6) Backend (Flask)**

### **6.1 Initialization (app.py)**

* Loads configuration from Config.

Creates an OpenAI client only if OPENAI\_API\_KEY exists:  
  
 client = OpenAI(api\_key=app.config["OPENAI\_API\_KEY"]) if app.config["OPENAI\_API\_KEY"] else None

* **Allowed models** are explicitly whitelisted:  
    
   ALLOWED\_MODELS = {"gpt-4o-mini", "gpt-4o", "gpt-4.1-mini"}
* If a request specifies a model not in ALLOWED\_MODELS, the app falls back to Config.OPENAI\_MODEL.

### **6.2 System Prompt**

SYSTEM\_PROMPT = '''

You are Meal Plan Specialist, ...

Return the plain text. Dont use markdowns.

'''

* This prompt is **not automatically injected** in /api/chat or /api/chat/stream. The client provides the messages list. (The frontend currently sends its own system message)
* The (disabled) meal plan endpoint uses this prompt.

### **6.3 Security Headers**

@app.after\_request def set\_security\_headers(resp):

* Content-Security-Policy (from config)
* X-Content-Type-Options: nosniff
* X-Frame-Options: DENY
* Referrer-Policy: strict-origin-when-cross-origin

### **6.4 Routes**

#### **6.4.1 GET /**

Renders templates/index.html via base.html.

* Injects:  
  + warn: If OPENAI\_API\_KEY is missing, displays a warning in the UI.
  + model: Shows the current server default model name.

#### **6.4.2 POST /api/chat (JSON, non-streaming)**

**Purpose:** One-shot chat completion, returns the entire reply.

**Request (JSON):**

{

"messages": [

{"role":"system","content":"..."},

{"role":"user","content":"Hello"}

],

"model": "gpt-4o-mini"

}

**Behavior:**

* Uses model if allowed; otherwise falls back to OPENAI\_MODEL.
* Calls client.chat.completions.create(...) with temperature and max\_tokens from config.

**Response (200):**

{ "content": "Assistant reply..." }

**Errors (500):**

{ "error": "Server missing OPENAI\_API\_KEY" }

or OpenAI error message.

The shipped UI does **not** call this endpoint; it uses the streaming endpoint below.

#### **6.4.3 POST /api/chat/stream (SSE streaming)**

**Server logic:**

* Creates a generator yielding SSE data: frames as chunks arrive.
* Catches exceptions and emits {"error": ...} frame.
* Always emits [DONE] terminator.

**cURL example (for testing SSE)**:

curl -N -X POST http://localhost:8080/api/chat/stream \

-H "Content-Type: application/json" \

-d '{"messages":[{"role":"system","content":"You are helpful."},{"role":"user","content":"Hi!"}]}'

#### **6.4.4 POST /api/mealplan (Structured JSON) — Disabled & malformed**

The code block under:

# Meal Plan (structured JSON)

# @app.post("/api/mealplan")

# def api\_mealplan():

if client is None:

...

has been partially commented. As written:

* The decorator **and** the def api\_mealplan(): line are commented out.
* The indented function body is **not** commented, leaving stray indented code at top level.
* **Result:** This will raise a **SyntaxError / IndentationError** if executed.

**What it is intended to do (when fixed):**

* Accepts preferences JSON (days, meals\_per\_day, calories, diet, cuisine, exclusions, budget, locale).
* Builds a **strict JSON schema** instruction string and sends it to OpenAI with response\_format={"type": "json\_object"}.
* Parses the model response with json.loads and returns it.

## **7) Frontend (Templates & JS)**

### **7.1 templates/base.html**

* Imports Tailwind via CDN.
* Wires static/css/custom.css and static/js/app.js.
* Sets a bright food-inspired gradient background.

Displays the configured model in the header:  
  
 Model: <span id="modelName">{{ model }}</span>

* Injects global APP\_WARN (if API key is missing), which the frontend prints in the chat.

### **7.2 templates/index.html**

* Renders a **single chat panel** centered vertically/horizontally.
* Controls:  
  + **Speak replies** toggle → enables TTS of assistant replies.
  + **Clear** button → clears local chat history and UI.
* Output is a <pre id="chatLog"> area that receives chat bubbles.
* Form with:  
  + **Mic button** (voice input via Web Speech API).
  + **Text input** (#chatInput).
  + **Send** button.

There is **no visible meal-plan form** in this template. The frontend JS contains optional handlers for a plan form if present, but it is not included in index.html.

### **7.3 static/js/app.js — Key Components**

#### **7.3.1 Local Storage**

* History key: mp\_history (array of {role, content} messages).
* store.load() / store.save(items).

#### **7.3.2 TTSStreamer (Text-to-Speech)**

A helper class to read assistant replies as they stream:

* Buffers incoming text and **speaks full sentences** (., !, ?, …) above a minChunk length (default 45 chars) to avoid choppy TTS.
* Cleans simple markdown symbols before speaking for more natural audio (\*\_#>~ and links).
* Public methods:  
  + ingest(delta: string): Append streamed text; speak full sentences when possible.
  + drain(): Force speaking whatever is currently buffered (used when toggled on mid-reply).
  + flush(): Speak remaining tail on stream end.
  + cancel(): Stop all speech and clear buffer.

Toggle control: #ttsToggle. When switched **on** mid-reply, drain() is called to speak accumulated text.

#### **7.3.3 Chat UI helpers**

* addMessage(role, content) creates chat bubbles:  
  + User → right aligned, colorful gradient bubble.
  + Assistant → left aligned, white bubble.

**Sanitization:**

* Assistant deltas are sanitized by replacing < and > before insertion.
* User messages replace < only. This is enough to prevent tag injection, as > alone won’t create HTML tags. If you want belt-and-suspenders, you can replace both on user messages too.

#### **7.3.4 streamChat(messages)**

* fetch('/api/chat/stream', ...) with the messages array.
* Parses the **SSE** stream:  
  + Creates a placeholder bubble (“Typing…”).
  + For each data: frame:  
    - If {"delta": "..."} → appends to bubble and feeds TTS.
    - If {"error": "..."} → renders error in red.
    - If [DONE] → calls ttsStreamer.flush() and finishes.

#### **7.3.5 Form submit handler**

* Cancels any ongoing TTS (to avoid feedback during dictation).
* Appends user message to local history and UI.

Builds messages sent to the server:  
  
 const messages = [

{ role: "system", content: "You are a helpful nutrition assistant." },

...history

];

* **Note:** This is a client-side system prompt and does **not** use the SYSTEM\_PROMPT defined on the server.
* Calls streamChat(messages).

#### **7.3.6 Clear Chat**

* Clears mp\_history and the chat log UI.
* Cancels any TTS.

#### **7.3.7 Voice Input (Web Speech API)**

* Uses window.SpeechRecognition || window.webkitSpeechRecognition.
* recognizer.start() / recognizer.stop() toggled by the mic button.
* Interim results populate the input field; final result persists on input.
* On start, TTS is cancelled to avoid the model’s voice being re-captured by the mic.

**Browser support:** Web Speech API is broadly supported in Chrome-based browsers; Safari support varies; Firefox lacks native SR.

#### **7.3.8 Meal Plan UI (optional; currently not present)**

* Code checks for #planForm/#planOut. If present, it would:  
  + POST to /api/mealplan.
  + Render a day-by-day plan with totals and meals.
  + Provide **Print** and **Download JSON** buttons.

Given the route is disabled and the form is not in index.html, this block is currently **inactive**.

## **8) Data Contracts**

### **8.1 Chat messages (OpenAI Chat Completions)**

Array of objects:

[

{"role":"system","content":"..."},

{"role":"user","content":"..."},

{"role":"assistant","content":"..."}

]

### **8.2 Meal Plan schema (intended)**

The frontend renderPlan() expects days, optional shopping\_list, and optional notes, with fields exactly as named.

## **9) Error Handling & Edge Cases**

* **Missing API key** → Both endpoints return {error: "Server missing OPENAI\_API\_KEY"} with status 500. The homepage also shows a warning bubble.
* **Model not allowed** → Fallback to Config.OPENAI\_MODEL silently.
* **OpenAI exceptions** → Returned as {error: "<message>"} with status 500.
* **Streaming parse errors** → Ignored in the client loop (bad chunks are skipped).
* **Meal plan endpoint** → Currently malformed; fix as per §6.4.4.

## **10) Security Notes**

* **CSP** is enabled and reasonably strict given CDN Tailwind.
* **XSS mitigation**: Assistant deltas and user text are escaped before insertion (at least < on both, </> on assistant). Consider escaping both chars for user text as well for consistency.
* **Frames disabled** (X-Frame-Options: DENY).
* **Referrer policy** is strict to limit leakage.

## **11) Deployment Notes**

* **SSE support**:  
  + If deploying behind **Nginx**, ensure:  
    - proxy\_buffering off;
    - X-Accel-Buffering: no respected by your proxy.
  + If using **Gunicorn**, prefer:  
    - Worker class that supports streaming (e.g., gevent or eventlet), or run with default sync but ensure no buffering at proxy layer.
* **HTTPS**: Use TLS in production to enable the Web Speech API reliably and to protect tokens.
* **Env injection**: Set OPENAI\_API\_KEY securely (secrets manager, env vars).

## **12) Known Mismatches & Improvements**

1. **System Prompt Source**
   * Server defines SYSTEM\_PROMPT (“Meal Plan Specialist … no markdown”).
   * Client uses its own system message (“helpful nutrition assistant”).
   * **If you want a single source of truth**, consider:  
     + Injecting the server’s SYSTEM\_PROMPT in /api/chat//api/chat/stream by **prepending** it to messages server-side, or
     + Exposing the server prompt to the client and always using that.
2. **Meal Plan Endpoint is Broken**
   * The partial comment leaves an indented block at top level → **SyntaxError**.
   * Either fully enable it or fully comment it.
3. **Markdown vs Plain Text**
   * SYSTEM\_PROMPT says “Return the plain text. Don’t use markdowns.”
   * The UI renders plain text (escaped) in bubbles; this is consistent.
   * If you want **Markdown rendering**, switch to a Markdown renderer on the client and remove the “no markdown” instruction.
4. **User Input Escaping**
   * Currently only < is escaped in user messages; consider escaping > too for consistency.
5. **Model Allowlist**
   * Keep ALLOWED\_MODELS in sync with available models. Provide a clear error if a requested model isn’t allowed instead of silently falling back (optional product choice).

## **13) API Usage Examples**

### **13.1 One-shot Chat**

curl -X POST http://localhost:8080/api/chat \

-H "Content-Type: application/json" \

-d '{

"model":"gpt-4o-mini",

"messages":[

{"role":"system","content":"You are a helpful nutrition assistant."},

{"role":"user","content":"Suggest a 500 kcal breakfast."}

]

}'

**Response:**

{ "content": "Greek yogurt parfait with berries..." }

### **13.2 Streaming Chat (SSE)**

Parse incoming data: events on the client.

## **14) Cross-Browser Considerations**

* **SSE** is widely supported; IE is not supported.
* **Web Speech API**:  
  + Speech Recognition: Chrome-based browsers best; Safari partial; Firefox lacks SR.
  + Speech Synthesis: Wider support, but voice availability varies by OS/browser.

## **15) Troubleshooting**

* **Nothing speaks**: Ensure “Speak replies” toggle is on, check browser permissions, and confirm window.speechSynthesis is present.
* **Mic not working**: Browser doesn’t support SpeechRecognition or permission denied.
* **No stream**: Proxy buffering; confirm X-Accel-Buffering: no and proxy\_buffering off.
* **500 with “Server missing OPENAI\_API\_KEY”**: Set OPENAI\_API\_KEY in .env.

## **16) Quick Reference (Functions & Responsibilities)**

* **Flask**
  + index() → renders main page.
  + api\_chat() → non-streaming chat completion.
  + api\_chat\_stream() → streaming chat via SSE.
  + api\_mealplan() → (intended) structured plan JSON (currently disabled).
* **Frontend**
  + TTSStreamer → sentence-aware streaming TTS.
  + streamChat() → handles SSE stream, updates UI, feeds TTS.
  + setupVoice() / mic click → voice input via Web Speech API.
  + addMessage() → bubble rendering.
  + renderPlan() → (optional) renders structured plan if endpoint/UI are enabled.