# **Smart Menu Planner & Meal Builder — Technical Process Document**

This document explains, end-to-end, how the system builds an ingredient inventory, generates menus (Main/Side/Soup/Dessert/Drink), recalculates calories and price using that inventory, and provides a Meal Builder UI that estimates grams, calories, and price in **yen**.

## **1) Repository layout**

mekyu\_up/

├── cvae\_recommender.py # menu generator (no heavy training required)

├── ui\_menu\_planner.py # Gradio UI: calendar + edit page + meal builder tab

├── inventory\_list.csv # normalized ingredient inventory (per-gram cal/cost in yen)

├── clustered\_dishes\_new.csv # dish library with DishName/Category/Calories/Price/Ingredients

├── syn\_dish.csv # raw source (used to bootstrap inventory)

└── README.md

## **2) Data assets & schemas**

### **2.1 clustered\_dishes\_new.csv (input to menu generator)**

Required columns:

* DishName (str)
* Category (str: one of main, side, soup, dessert, drinks)
* Ingredients (str: comma-separated with optional grams, e.g., salmon (80g), soy sauce (5g))
* Calories (float) — dish-level total
* Price (float) — dish-level total **in yen**

### **2.2 syn\_dish.csv (optional source to build inventory)**

* Contains Ingredients column in the same format as above.
* Used once to create inventory\_list.csv.

### **2.3 inventory\_list.csv (canonical inventory; used by UI)**

Columns:

* Ingredient (lowercase, canonical name; e.g., salmon, soy sauce)
* Avg\_Grams (float; optional reference size from your dataset)
* Calories\_per\_g (float; kcal per gram)
* Cost\_per\_g (float; **yen per gram**)

**Invariant:** All costs in the app assume **yen**. Ensure Cost\_per\_g is in JPY.

## **3) Building the inventory (from syn\_dish.csv)**

This process parses raw ingredient strings, aggregates grams, and assigns per-gram calories and **yen** cost. It outputs inventory\_list.csv.

### **Steps**

1. **Parse ingredients & grams**
   * Regex: (.+?)\s\*\(([\d\.]+)g\)
   * For items without a weight, assign a small default (e.g., 1g) just for averaging.
2. **Aggregate across dishes**
   * For each ingredient name (lowercased, trimmed), sum total grams and count occurrences.
   * Compute Avg\_Grams = total\_grams / count.
3. **Assign per-gram calories and cost**
   * Use heuristic mappings (meat/fish/oil/rice/etc.).
   * **Currency:** assign **yen per gram** directly (recommended), or convert from USD using a fixed rate.
4. Example assignment (JPY):  
   * Chicken: cal\_per\_g=2.0, cost\_per\_g=3.0 (≈ ¥3/g)
   * Rice: cal\_per\_g=1.3, cost\_per\_g=1.0
   * Oil/Butter: cal\_per\_g=9.0, cost\_per\_g=4.5
   * Vegetables: cal\_per\_g=0.4, cost\_per\_g=1.0
   * Sauces: cal\_per\_g=2.0, cost\_per\_g=2.0
   * Sugar: cal\_per\_g=4.0, cost\_per\_g=1.0
   * Milk/Yogurt: cal\_per\_g=0.6, cost\_per\_g=2.0
   * Fruits/Tomato: cal\_per\_g=0.5, cost\_per\_g=2.0
5. You can refine these values later with vendor pricing.
6. **Normalize names**
   * Lowercase names.
   * Map synonyms (e.g., steamed rice → rice, soy sauce broth → soy sauce) if desired, to improve matching.
7. **Write inventory\_list.csv**
   * One row per canonical ingredient with Calories\_per\_g and Cost\_per\_g in **yen**.

## **4) Menu generation (cvae\_recommender.py)**

This module exposes a function the UI imports:

### **Public API**

* generate\_month\_menu\_with\_meals(days: int) -> pd.DataFrame

### **Process**

1. **Load clustered\_dishes\_new.csv** and validate required columns.
2. **Category dictionaries**
   * Pre-split dishes by category: main, side, soup, dessert, drinks.
3. **Pickers with “no repeat”**
   * For each meal (breakfast, lunch, dinner), we track a short “recent” list per category to reduce repetition.
   * If a category list is empty, the function raises a helpful exception to surface missing data early.
4. **Day menu assembly**
   * For each meal, pick one Main, Side, Soup, Dessert, Drink.
   * Store **flat columns per field**, e.g.:  
     + breakfast\_Main, breakfast\_Main\_Ingredients, breakfast\_Main\_Calories, breakfast\_Main\_Price, etc.
   * **Important:** Keeping these as flat columns (strings/floats) avoids the “str has no attribute get” error you saw previously.
5. **Return as DataFrame**
   * One row per day, with Day column and all meal/category columns.

## **5) UI logic (ui\_menu\_planner.py)**

This Gradio UI has **three tabs**:

1. **Menu Generation (calendar view)**
2. **Edit Ingredients for generated menus**
3. **Meal Builder (manual ingredient → totals, in yen)**

### **5.1 Load inventory**

inventory\_df = pd.read\_csv("inventory\_list.csv")

inventory\_df["Ingredient"] = inventory\_df["Ingredient"].str.lower()

inventory\_df.set\_index("Ingredient", inplace=True)

### **5.2 Calorie/Cost recomputation**

* **Function:** recalc\_nutrition(ingredient\_list: str) -> (calories: float, price\_yen: float)
* **Input:** comma-separated items, each optionally with grams in parentheses: name (80g)
* **Steps:**
  1. Split by comma.
  2. Regex extract (name, grams). Default grams if missing (e.g., 50g).
  3. Lowercase name, look up in inventory\_df.
  4. Accumulate: total\_cal += cal\_per\_g \* grams
  5. Accumulate: total\_cost += cost\_per\_g \* grams (**yen**)
* **Output:** totals in kcal and **yen**.

This keeps the UI independent of dish-level totals and ensures edits reflect per-ingredient costs accurately.

### **5.3 Menu Generation tab**

* Inputs:  
  + days slider (1–30)
  + Calorie/Cost range sliders (optional; can be wired into future scoring)
* Button: “Generate Menus”
* Action:  
  + Calls generate\_month\_menu\_with\_meals(days)
  + Stores the DataFrame in generated\_menus
  + Renders calendar using flat columns:  
    - For each meal, show Main + Side + Soup + Dessert + Drink
    - Sum per-component calories and price columns for the totals displayed.
    - Show an expandable <details> block listing all ingredients.

### **5.4 Edit Ingredients tab**

* Inputs: Day, Meal (breakfast|lunch|dinner)
* “Show Current Ingredients”: reads f"{meal}\_Main\_Ingredients" and formats a preview with recalculated totals via recalc\_nutrition().
* “Update Ingredient”:  
  + Adds/removes an ingredient string in the **Main** list only (by design; can be extended to other categories).
  + After edit, calls recalc\_nutrition() and updates **Main** calories and price columns:  
    - f"{meal}\_Main\_Calories"
    - f"{meal}\_Main\_Price"
  + Re-renders the calendar.

**Design note:** We edit only the Main’s ingredients in this tab to keep UX simple; additional radios can target Side/Soup/Dessert/Drink as needed.

### **5.5 Meal Builder tab (from inventory only)**

* Purpose: Manually build a meal by selecting ingredients from inventory\_list.csv and specifying grams, then compute totals.
* UI:  
  + A fixed number of rows (e.g., 10) to avoid list-of-lists wiring issues.
  + For each row:  
    - Dropdown: Ingredient (choices from inventory\_df.index)
    - Number input: grams (default 0)
  + “Compute totals” button
  + Output: a text box (or HTML) with:  
    - Line items: name — grams g — kcal — ¥cost
    - Totals: Total grams, Total kcal, Total cost (¥)
* Compute:  
  + For each non-empty row with grams > 0:  
    - kcal = Calories\_per\_g \* grams
    - yen = Cost\_per\_g \* grams
  + Sum down the column.

**Currency**: The Meal Builder assumes Cost\_per\_g is already **yen**. It displays ¥ and rounds to whole yen for readability.

## **6) Error handling & common pitfalls**

1. **'int' object has no attribute '\_id' in Gradio** Caused by passing raw constants instead of components to .click.  
    **Fix:** Ensure .click(inputs=[...]) only contains Gradio components (e.g., sliders), not scalar literals.
2. **'a' cannot be empty unless no samples are taken** Occurs when a category list (e.g., soup) is empty in clustered\_dishes\_new.csv.  
    **Fix:** Populate all required categories or guard with a check that raises a clear error showing the missing category.
3. **'str' object has no attribute 'get'** Caused by storing dicts in DataFrame cells and later treating them as dicts/string inconsistently.  
    **Fix:** Use **flat columns** for each meal/category field (\*\_Ingredients, \*\_Calories, \*\_Price) and keep them as primitives.
4. **Dropdown “value not in choices” warning** Happens if default value not in inventory index.  
    **Fix:** leave blank default (value=None) or enable allow\_custom\_value=True.
5. **Meal Builder: “takes N args but M were given”** Caused by passing a list of components as a single list input.  
    **Fix:** pass the flattened list of inputs; in callback, reconstruct pairs by slicing.
6. **Prices off by currency** If totals look “too small” or “too big”, confirm Cost\_per\_g units:  
   * Must be **yen per gram**.
   * If converting from USD, multiply by an exchange rate (e.g., USDJPY = 150.0) and store yen in the CSV.

## **7) Performance notes**

* The current generator doesn’t require heavy model training; it samples dishes by category with simple “no recent repeat” logic.
* If you later switch to a trained CVAE:  
  + Reduce epochs for faster iterations (e.g., 50–100).
  + Use CPU-only unless you have GPU acceleration.
  + Save/load the model to avoid retraining each run.

## **8) Validation & testing**

1. **Inventory spot checks**
   * Pick 5 ingredients; verify Cost\_per\_g and Calories\_per\_g are realistic.
   * Confirm names match those used in dish Ingredients.
2. **Menu generation**
   * Run for days=3; confirm all meals have 5 components and non-empty ingredient strings.
   * Ensure no category is missing.
3. **Edit ingredient**
   * Add soy sauce (5g) to a Main dish; confirm totals increase by 5 \* Cost\_per\_g['soy sauce'] yen and 5 \* Calories\_per\_g['soy sauce'] kcal.
4. **Meal Builder**
   * Build a meal with known values (e.g., rice 150g, chicken 100g); hand-calc and compare with UI totals.
   * Confirm totals display ¥ and are rounded to integers.

## **9) Deployment & run**

* Create/refresh inventory\_list.csv (if building from syn\_dish.csv).
* Ensure clustered\_dishes\_new.csv includes all categories.
* Start the UI:

python ui\_menu\_planner.py

* Open the provided local URL in a browser.

## **10) Appendix**

### **A) Regex notes**

* Ingredient parsing:  
  + Pattern: (.+?)\s\*\(([\d\.]+)g\)
  + Captures name and gram value.
  + Whitespace and case normalized to lower case.
* UI recalc fallback:  
  + If no grams provided (e.g., soy sauce), the UI uses a default (e.g., 50g) to prevent zero-valued entries.

### **B) Column naming (flat schema)**

For each meal m ∈ {breakfast,lunch,dinner} and category c ∈ {Main,Side,Soup,Dessert,Drink}, the DataFrame has:

* m\_c — dish name (str)
* m\_c\_Ingredients — dish ingredients (str)
* m\_c\_Calories — numeric total kcal (float)
* m\_c\_Price — numeric price **yen** (float)

Totals shown in the calendar are computed as the sum of the five categories’ calories and prices per meal.

## **11) Checklist**

* inventory\_list.csv is in **yen** per gram.
* All five categories exist in clustered\_dishes\_new.csv.
* UI shows ¥ and rounds currency to whole yen.
* Editing Main ingredients updates \*\_Main\_Calories and \*\_Main\_Price.
* Meal Builder computes from inventory only (no CVAE), using per-gram values