# **Recommender & CP-SAT Design — From Synthetic Data to UI Output**

This document zooms in on the **algorithmic pipeline**—how we build data, learn light embeddings with a CVAE, and formulate a **constraint programming (CP-SAT)** optimization (via **Google OR-Tools**) to pick daily menus (Main/Side/Soup/Dessert/Drink) that meet calorie and cost targets while reducing repeats. It ends with how the UI consumes the results.

## **1) Data preparation**

### **1.1 Synthetic dish library (if you don’t have a real one yet)**

Target schema (clustered\_dishes\_new.csv):

| **Column** | **Type** | **Notes** |
| --- | --- | --- |
| DishName | str | e.g., "Teriyaki Chicken Bowl" |
| Category | str | one of main, side, soup, dessert, drinks |
| Ingredients | str | comma‐sep, each optionally with grams, e.g., "chicken (120g), rice (150g)" |
| Calories | float | dish total kcal |
| Price | float | dish price **in yen** |

**Minimal synthetic generator (example):**

# create\_synthetic\_dishes.py

import numpy as np, pandas as pd, random

rng = np.random.default\_rng(7)

cats = ["main", "side", "soup", "dessert", "drinks"]

def make\_dish(cat, i):

base = {

"main": dict(cal=(450, 800), price=(250, 600)),

"side": dict(cal=(150, 350), price=(80, 200)),

"soup": dict(cal=(80, 220), price=(60, 150)),

"dessert":dict(cal=(180, 450), price=(100, 300)),

"drinks": dict(cal=(0, 200), price=(50, 150)),

}[cat]

cal = float(rng.integers(\*base["cal"]))

price = float(rng.integers(\*base["price"]))

ings = ["chicken (120g)", "rice (150g)", "soy sauce (10g)", "carrot (60g)", "sugar (8g)"]

chosen = ", ".join(random.sample(ings, k=rng.integers(2,4)))

return {

"DishName": f"{cat.title()} #{i}",

"Category": cat,

"Ingredients": chosen,

"Calories": cal,

"Price": price,

}

rows=[]

for cat in cats:

for i in range(80): # ~80 per category

rows.append(make\_dish(cat, i+1))

pd.DataFrame(rows).to\_csv("clustered\_dishes\_new.csv", index=False)

Swap with your real library when ready; just keep the same columns.

## **2) Light CVAE embeddings**

We use a tiny **CVAE** to learn a compact latent vector per dish (from [Calories, Price] or a richer feature set if you have it). The latent can be used to:

* encourage **diversity** (choose dishes with different embeddings),
* interpolate to **targets** (calories/cost) during candidate pre-filtering.

**Key idea:** CVAE is not generating menus; it only produces an embedding that CP-SAT can reference indirectly (e.g., by preferring distance from the last chosen main).

## **3) CP-SAT formulation (Google OR-Tools)**

We cast menu selection as a **0/1 integer program**.

### **3.1 Indices**

* Days d ∈ {1..D}
* Meals m ∈ {breakfast, lunch, dinner}
* Categories c ∈ {Main, Side, Soup, Dessert, Drink}
* Items within a category i ∈ S\_{c} (candidate dishes for that category)

### **3.2 Decision variables**

* x[d,m,c,i] ∈ {0,1} — pick dish i of category c for meal m on day d.

### **3.3 Parameters**

* cal\_i, price\_i — from dish library (kcal, yen)
* target\_cal[m] ∈ [L\_cal[m], U\_cal[m]] — calorie range per meal
* target\_cost[m] ∈ [L\_cost[m], U\_cost[m]] — yen range per meal
* Optionally, recent\_set[m,c] — last K dishes to avoid

### **3.4 Constraints**

**Exactly one per category** (for each day & meal):  
  
 ∑\_{i ∈ S\_c} x[d,m,c,i] = 1

**Calorie window** (per day & meal):  
  
 L\_cal[m] ≤ ∑\_{c} ∑\_{i} cal\_i \* x[d,m,c,i] ≤ U\_cal[m]

**Cost window** (per day & meal):  
  
 L\_cost[m] ≤ ∑\_{c} ∑\_{i} price\_i \* x[d,m,c,i] ≤ U\_cost[m]

1. **No recent repeats** (soft or hard). Easiest: **remove** items in recent\_set from candidate pools before building the model. (Hard form is also possible via ∑ x[...] ≤ 0 for disallowed items.)

### **3.5 Objective (examples)**

* **Minimize deviation from midpoints**:  
  + Let C\_mid[m] = (L\_cal[m]+U\_cal[m])/2, P\_mid[m] = (L\_cost[m]+U\_cost[m])/2.

Introduce nonnegative slack variables cal\_pos[d,m], cal\_neg[d,m], same for price, and constrain:  
  
 ∑ cal\_i x - C\_mid[m] = cal\_pos[d,m] - cal\_neg[d,m]

∑ price\_i x - P\_mid[m] = price\_pos[d,m] - price\_neg[d,m]

* + Objective: minimize Σ (cal\_pos + cal\_neg + price\_pos + price\_neg)
* Or **Minimize total cost** subject to staying inside calorie windows.
* Or **Maximize variety** via penalties for reusing ingredients or selecting mains close in CVAE latent space (approximate with a precomputed “similarity penalty” per dish and add penalty\_i \* x to the objective).

CP-SAT solves this mixed boolean linear program very fast for small candidate sets (e.g., 10–30 dishes per category per meal). For large libraries, build a **candidate pool** per day/meal/category first (random sample or top-K near the target calories).

## **4) Reference implementation (ready to import)**

### **cvae\_recommender.py (CP-SAT selector with optional CVAE embeddings)**

# cvae\_recommender.py

import numpy as np

import pandas as pd

from ortools.sat.python import cp\_model

# ======================

# Load dish library

# ======================

df = pd.read\_csv("clustered\_dishes\_new.csv")

REQUIRED = {"DishName","Category","Calories","Price","Ingredients"}

missing = REQUIRED - set(df.columns)

if missing:

raise ValueError(f"clustered\_dishes\_new.csv missing columns: {missing}")

# Normalize categories

df["Category"] = df["Category"].str.lower().str.strip()

VALID\_CATS = ["main","side","soup","dessert","drinks"]

# Keep only valid categories; warn if any are empty

cat\_buckets = {c: df[df["Category"]==c].reset\_index(drop=True) for c in VALID\_CATS}

for c, sub in cat\_buckets.items():

if sub.empty:

raise ValueError(f"No dishes found for category '{c}'.")

# ======================

# Defaults (can be overridden by UI)

# ======================

DEFAULT\_CAL\_RANGES = {

"breakfast": (300, 500),

"lunch": (600, 900),

"dinner": (500, 800),

}

DEFAULT\_COST\_RANGES = {

"breakfast": (150, 400), # yen

"lunch": (300, 700),

"dinner": (250, 600),

}

MEALS = ["breakfast","lunch","dinner"]

CATS = ["Main","Side","Soup","Dessert","Drink"] # capitalized for output columns

# ======================

# Helper: build candidate pools (limit size for speed)

# ======================

def candidate\_pool(cat\_df, pool\_size=25, seed=None):

if len(cat\_df) <= pool\_size:

return cat\_df.index.to\_list()

rng = np.random.default\_rng(seed)

return rng.choice(cat\_df.index.to\_list(), size=pool\_size, replace=False).tolist()

# ======================

# Solve one (day, meal) via CP-SAT

# ======================

def solve\_meal(cat\_buckets, cal\_range, cost\_range, pool\_size=25, seed=None, time\_limit\_s=2.0):

model = cp\_model.CpModel()

rng = np.random.default\_rng(seed)

# Candidate indices per category

pools = {}

for c in VALID\_CATS:

pools[c] = candidate\_pool(cat\_buckets[c], pool\_size=pool\_size, seed=rng.integers(1e9))

# Decision variables x[c,i] \in {0,1}

x = {}

for c in VALID\_CATS:

for i in pools[c]:

x[(c,i)] = model.NewBoolVar(f"x\_{c}\_{i}")

# Exactly one per category

for c in VALID\_CATS:

model.Add(sum(x[(c,i)] for i in pools[c]) == 1)

# Totals

cal\_total = sum(int(cat\_buckets[c].loc[i,"Calories"]) \* x[(c,i)] for c in VALID\_CATS for i in pools[c])

price\_total = sum(int(cat\_buckets[c].loc[i,"Price"]) \* x[(c,i)] for c in VALID\_CATS for i in pools[c])

# Windows

Lc, Uc = cal\_range

Lp, Up = cost\_range

model.Add(cal\_total >= int(Lc))

model.Add(cal\_total <= int(Uc))

model.Add(price\_total >= int(Lp))

model.Add(price\_total <= int(Up))

# Objective: minimize deviation from midpoints (linear, with slacks)

cal\_mid = int((Lc+Uc)//2)

price\_mid = int((Lp+Up)//2)

cal\_dev\_pos = model.NewIntVar(0, int(1e6), "cal\_pos")

cal\_dev\_neg = model.NewIntVar(0, int(1e6), "cal\_neg")

price\_dev\_pos = model.NewIntVar(0, int(1e6), "price\_pos")

price\_dev\_neg = model.NewIntVar(0, int(1e6), "price\_neg")

model.Add(cal\_total - cal\_mid == cal\_dev\_pos - cal\_dev\_neg)

model.Add(price\_total - price\_mid == price\_dev\_pos - price\_dev\_neg)

model.Minimize(cal\_dev\_pos + cal\_dev\_neg + price\_dev\_pos + price\_dev\_neg)

# Solve

solver = cp\_model.CpSolver()

solver.parameters.max\_time\_in\_seconds = float(time\_limit\_s)

solver.parameters.num\_search\_workers = 8 # parallel search

status = solver.Solve(model)

if status not in (cp\_model.OPTIMAL, cp\_model.FEASIBLE):

return None # infeasible; caller will retry with wider pools or ranges

# Extract chosen rows per category

chosen = {}

for c in VALID\_CATS:

for i in pools[c]:

if solver.Value(x[(c,i)]) == 1:

row = cat\_buckets[c].loc[i]

chosen[c] = {

"name": row["DishName"],

"ingredients": row["Ingredients"],

"cal": float(row["Calories"]),

"price": float(row["Price"]),

}

break

# Totals

totals = {

"cal": sum(chosen[c]["cal"] for c in VALID\_CATS),

"price": sum(chosen[c]["price"] for c in VALID\_CATS),

}

return chosen, totals

# ======================

# Public API

# ======================

def generate\_month\_menu\_with\_meals(

days=7,

cal\_ranges=None,

cost\_ranges=None,

pool\_size=25,

solver\_time\_limit\_s=2.0,

seed=42,

):

"""

Returns a flat DataFrame with columns:

'{meal}\_{Cat}', '{meal}\_{Cat}\_Ingredients', '{meal}\_{Cat}\_Calories', '{meal}\_{Cat}\_Price' (Cat in Main/Side/Soup/Dessert/Drink)

'Day'

"""

rng = np.random.default\_rng(seed)

cal\_ranges = cal\_ranges or DEFAULT\_CAL\_RANGES

cost\_ranges = cost\_ranges or DEFAULT\_COST\_RANGES

rows = []

for day in range(1, days+1):

record = {"Day": day}

for meal in MEALS:

result = solve\_meal(

cat\_buckets,

cal\_range=cal\_ranges[meal],

cost\_range=cost\_ranges[meal],

pool\_size=pool\_size,

seed=rng.integers(1e9),

time\_limit\_s=solver\_time\_limit\_s,

)

# Simple backoff if infeasible: expand pool and retry once

if result is None:

result = solve\_meal(

cat\_buckets,

cal\_range=cal\_ranges[meal],

cost\_range=cost\_ranges[meal],

pool\_size=min(pool\_size\*2, 80),

seed=rng.integers(1e9),

time\_limit\_s=solver\_time\_limit\_s\*1.5,

)

if result is None:

# As a last resort, random pick (should be rare if ranges are reasonable)

for c in VALID\_CATS:

r = cat\_buckets[c].sample(1, random\_state=int(rng.integers(1e9))).iloc[0]

record[f"{meal}\_{c.title()}"] = r["DishName"]

record[f"{meal}\_{c.title()}\_Ingredients"] = r["Ingredients"]

record[f"{meal}\_{c.title()}\_Calories"] = float(r["Calories"])

record[f"{meal}\_{c.title()}\_Price"] = float(r["Price"])

continue

chosen, \_tot = result

for c in VALID\_CATS:

C = c.title()

record[f"{meal}\_{C}"] = chosen[c]["name"]

record[f"{meal}\_{C}\_Ingredients"] = chosen[c]["ingredients"]

record[f"{meal}\_{C}\_Calories"] = chosen[c]["cal"]

record[f"{meal}\_{C}\_Price"] = chosen[c]["price"]

rows.append(record)

return pd.DataFrame(rows)

**Notes**

* The solver runs **per (day, meal)**, selecting exactly one dish per category (5 picks total).
* It enforces **calorie and cost windows** and minimizes deviation from the midpoint.
* If infeasible with the default small pool, it retries with a larger pool once, then falls back to a random pick (very rare if ranges are sane).
* The output schema is **flat** and matches what your UI expects.

## **5) Putting it together (pipeline recap)**

1. **Data**
   * Prepare clustered\_dishes\_new.csv with at least a few dozen entries per category.
   * Costs are **yen** (not USD).
2. **(Optional) CVAE**
   * Train a tiny CVAE on numeric dish features to get latent embeddings if you want variety terms later. (This step is optional in the current CP-SAT code above.)
3. **Optimization (CP-SAT)**
   * For each day & meal, build a **small candidate pool** per category.
   * Add constraints for **one per category**, **calorie window**, **cost window**.
   * Objective minimizes deviation from midpoints for calories & cost.
   * (Optional) Augment objective with diversity penalties later.
4. **UI consumption**
   * UI calls generate\_month\_menu\_with\_meals(days=...).
   * Renders breakfast\_\*, lunch\_\*, dinner\_\* blocks with 5 components each, showing totals and expandable ingredients.
5. **Editing & Meal Builder**
   * The **Edit** tab recalculates per-dish totals from inventory\_list.csv (yen per gram), so edits reflect true cost/kcal.
   * The **Meal Builder** tab builds a meal purely from the inventory and computes totals in **yen**.

## **6) Extending the objective (optional ideas)**

* **Diversity via CVAE**: precompute a “similarity penalty” pen\_i vs. yesterday’s main based on latent distance. Add Σ pen\_i x[...] to the objective.
* **Inventory usage**: bias toward items that use existing stock—subtract a small reward for ingredients in surplus.
* **User constraints**: add hard bans (e.g., allergens), cuisine filters, or min/max counts per category over the week.

## **7) Performance tips**

* Keep pool sizes modest (10–30 per category) → sub-second solves per meal.
* Set max\_time\_in\_seconds to a small value (1–2s) to guarantee UI snappiness.
* If you see infeasibility, slightly **widen** calorie/cost windows or increase pool size.

## **8) What the UI shows**

* For each **day**:  
  + **Breakfast / Lunch / Dinner** rows with:  
     Main + Side + Soup + Dessert + Drink
  + A small line with **Calories** and **Price (¥)** totals per meal.
  + An expandable **Ingredients** list aggregating all five components.

The UI then supports editing, which recomputes the **Main** dish’s calories and price from inventory\_list.csv, and a **Meal Builder** page for manual assembly in **yen**.

### **TL;DR**

* **Data** → clean per dish (kcal/yen), per category.
* **(Optional) CVAE** → embeddings for future diversity.
* **CP-SAT** → exact one per category, calorie/cost windows, minimize deviation from midpoints.
* **UI** → renders flat columns; edit + meal builder compute costs using **yen per gram** inventory.