Shopping List Amalgamator – Optimal Plan

This document describes the recommended architecture and data flow for generating a weekly shopping list from meal plans in a way that avoids duplicate items (e.g. multiple salts/oils) and supports clean nutrition, scaling, and future AI-generated recipes.

1. Goals

- Aggregate ingredients from a weekly meal plan into a concise, human-friendly shopping list.
- Avoid duplicates like multiple entries for salt/pepper/oil.
- Support recipe scaling and nutrition calculations with simple arithmetic.
- Work for 1) existing messy recipes and 2) future AI-generated recipes.
- Keep the shopping list logic itself as simple and reliable as possible.

2. Core Data Model

2.1 ingredients
Each distinct thing you buy is stored once in an ingredients table:
- id: primary key
- name: e.g. 'chicken breast', 'rice', 'salt'
- category: e.g. 'meat', 'grains', 'spices', 'pantry'
- canonical_unit: base unit for maths, e.g. 'g', 'ml', 'piece'
- aggregation_mode: 'sum' for quantitative items, 'presence' for pantry staples
- grams_per_unit / ml_per_unit / pieces_per_unit: optional per-portion mappings

2.2 recipe_ingredients
Each recipe ingredient is stored in a canonical shape:
- recipe_id: which recipe it belongs to
- ingredient_id: FK to ingredients
- amount_canonical: number, always in canonical_unit
- canonical_unit: copied from ingredients (for clarity)
- display_text: the human-friendly text, e.g. '1 chicken breast (about 150g)'
- quality: 'auto', 'suspect', or 'manual' (for auditing normalisation)

3. Canonical vs Display

The key design is to separate maths from presentation:
- Canonical fields (ingredient_id, amount_canonical, canonical_unit) are used for:
  * shopping list aggregation
  * nutrition calculations
  * recipe scaling
- display_text is used wherever humans read the recipe.

Example:
  '1 chicken breast (about 150g)' is stored as:
    ingredient_id = chicken_breast
    amount_canonical = 150
    canonical_unit = 'g'
    display_text = '1 chicken breast (about 150g)'

4. Aggregation Logic (Amalgamator)

To build a weekly shopping list from a meal plan:

Step 1: gather recipe ingredients
- From meal_plans and plan_meals, get the set of recipe_ids in the chosen plan.

- Select all recipe_ingredients for those recipes, joined to ingredients.

Step 2: aggregate by ingredient_id
- Group by ingredient_id (never by raw text).
- For each group, compute SUM(amount_canonical) as total_amount.

Step 3: respect aggregation_mode
- If aggregation_mode = 'sum':
  * Keep total_amount in canonical_unit (e.g. 750g chicken, 400g rice).
- If aggregation_mode = 'presence':
  * Ignore total_amount and just mark that the ingredient is needed once (e.g. 'Salt').

This fixes problems like seven separate 'salt' entries and ensures '1 chicken breast' and '100g chicken' both contribute to a single aggregated total for chicken.

5. Conforming / Normalisation Pipeline

The conformer is a dedicated step that converts messy ingredient inputs into canonical form.
It is run once per recipe (on creation or migration), not on every request.

Input per ingredient (raw):
- text: '1 chicken breast'
- name: 'chicken breast' (optional pre-parse)
- quantity: 1
- unit: 'breast' or 'portion'

Output per ingredient (canonical):
- ingredient_id: FK to ingredients (e.g. chicken_breast)
- amount_canonical: numeric in canonical_unit (e.g. 150)
- canonical_unit: e.g. 'g'
- display_text: original human-facing string
- quality: auto/suspect/manual

Implementation suggestions:
- For legacy recipes, use an AI-assisted transformer plus validation to propose canonical forms.
- For new recipes, either:
  * constrain creation via a structured editor (choose ingredient + quantity + unit), or
  * force AI to output structured JSON matching your schema.
- Always keep the raw representation somewhere so you can re-run a better conformer in future.

6. Shopping List Formatting

Once you have aggregated totals per ingredient, format them for humans:

- For quantitative items (aggregation_mode = 'sum'):
  * Use total_amount and ingredient-specific rules to produce:
    e.g. 'Chicken breast – 4 pieces (~600g)', 'Rice – ~400g'.
  * These rules use grams_per_unit / ml_per_unit from the ingredient metadata.

- For pantry staples (aggregation_mode = 'presence'):
  * Show each only once without an exact amount:
    e.g. 'Salt', 'Black pepper', 'Olive oil'.

This layer is purely presentation: the amalgamator itself only works with ingredient_id and amount_canonical.

7. Nutrition and Scaling

Because every ingredient has a canonical amount, both nutrition and scaling are simple:

- Nutrition:
  * Store per-gram (or per-ml/piece) macros on ingredients.
  * Per recipe: sum(amount_canonical * macros_per_unit) per ingredient.
  * Per plan: sum over all recipes in the plan.

- Scaling:
  * For a scale factor f (e.g. 2x servings):
    new_amount_canonical = amount_canonical * f
  * Recompute display_text if desired (e.g. '3 chicken breasts (~450g)').

No special cases are needed in the amalgamator beyond simple multiplication and summation.

8. Implementation Checklist

1) Define and migrate to the canonical schema:
   - ingredients with canonical_unit and aggregation_mode.
   - recipe_ingredients with ingredient_id, amount_canonical, canonical_unit, display_text.

2) Build the conformer:
   - For legacy recipes: batch normalisation using AI + validation.
   - For new recipes: structured creation (editor or AI JSON) that always computes amount_canonical.

3) Implement amalgamator query:
   - Aggregate recipe_ingredients by ingredient_id for a given plan, respecting aggregation_mode.

4) Implement formatting rules per ingredient:
   - Decide how to show counts, approximate weights, and packs in the shopping list.

5) Wire into the Shop view:
   - Backend endpoint: GET /shop returns categories, items, and formatted labels.
   - Frontend renders the grouped list; no extra logic about amounts is needed on the client.