Okay, I understand. I will provide a platform-agnostic explanation of your app's business logic and data flow, as if it were re-implemented using the technologies you specified.

Here's the breakdown:

A. Authentication & User Context (Supabase Auth)

- **User Registration & Login**: Users register with email/password. Upon successful login, a JSON Web Token (JWT) is issued.
- **Session Management**: The JWT is stored client-side (e.g., in `localStorage`) and included in an `Authorization` header (`Bearer <JWT>`) with every subsequent request to the backend.
- **User Identification**: On each backend operation, the JWT is validated. The token contains a `user_id` and `email` claim, which uniquely identifies the user performing the operation. This ID is used to fetch user-specific data from a `users` table.
- **Initial User Provisioning**: When a user first logs in, their basic profile is automatically created if it doesn't exist, including default settings like preferred currency, subscription tier (e.g., 'free'), and initial usage counters.
- **Household Association**: Users are linked to a `households` table via a `household_id`. This allows for shared data within a household.

B. Authorisation & Access Control (Supabase Row Level Security - RLS)

Access to data is strictly controlled based on user identity and household membership:

- **`users` table**:
- **SELECT**: A user can read their own record (`auth.uid() = id`). Admin users can read all user records (`auth.role() = 'admin'`).
- **UPDATE**: A user can update their own record (`auth.uid() = id`). Admin users can update all user records (`auth.role() = 'admin'`).

- **DELETE**: Only admin users can delete user records (`auth.role() = 'admin'`).
- **`receipts` table**:
- **SELECT**: Users can read receipts belonging to their household (`household_id = (SELECT household_id FROM users WHERE id = auth.uid())`). Admin users can read all receipts (`auth.role() = 'admin'`).
- **INSERT/UPDATE/DELETE**: Users can modify receipts belonging to their household (`household_id = (SELECT household_id FROM users WHERE id = auth.uid())`). Admin users can modify all receipts (`auth.role() = 'admin'`).
- **`budgets` table**:
- **SELECT/INSERT/UPDATE/DELETE**: Similar to `receipts`, access is granted if `household_id` matches the user's `household_id`. Admin users have full access.
- **`correction_logs`, `failed_scan_logs`, `nutrition_facts`, `failed_nutrition_lookups` tables**:
- **SELECT/INSERT/UPDATE/DELETE**: Access is granted if `household_id` matches the user's `household_id`. Admin users have full access.
- **`ocr_feedback` table**:
- **SELECT/INSERT/UPDATE/DELETE**: Access is granted if `created_by` matches the user's email. Admin users have full access.
- **`households` table**:
- **SELECT**: Users can read their associated household record (`id = (SELECT household_id FROM users WHERE id = auth.uid())`). Admin users can read all households.
- **INSERT/UPDATE/DELETE**: Only the `admin_id` of the household can modify or delete the household record (`admin_id = auth.uid()`). Admin users have full access.
- **`household_invitations` table**:
- **SELECT**: Users can read invitations they've sent or received (`household_id = (SELECT household_id FROM users WHERE id = auth.uid()) OR invitee_email = auth.email()`). Admin users have full access.

- **INSERT/UPDATE/DELETE**: Users can create/manage invitations for their household (`household_id = (SELECT household_id FROM users WHERE id = auth.uid())`). Admin users have full access.
- **`credit_logs` table**:
- **SELECT/INSERT/UPDATE/DELETE**: Access is granted if `household_id` matches the user's `household_id`. Admin users have full access.
- **`recipes` table**:
- **SELECT**: Curated recipes are publicly readable. User-parsed recipes are readable if `household_id` matches the user's `household_id` or if `created_by` matches the user's email (for personal, un-householded recipes). Admin users can read all recipes.
- **INSERT/UPDATE/DELETE**: User-parsed recipes can be modified if `household_id` matches the user's `household_id` or if `created_by` matches the user's email. Admin users have full access.
- **`ingredient_maps`, `grocery_categories`, `official_category_mappings`, `inflation_data` tables**:
- **SELECT**: All users can read these (public data).
- **INSERT/UPDATE/DELETE**: Only admin users can modify these tables.
- **`test_runs`, `ocr_quality_logs` tables**:
- **SELECT/INSERT/UPDATE/DELETE**: Only admin users can access these tables.

C. Receipt Upload & Storage

- **Upload Trigger**: A user initiates a receipt upload, typically from a "Scan Receipt" interface, providing one or more image files.
- **Temporary Storage**: The image files are first uploaded to a secure, temporary cloud storage bucket (e.g., AWS S3).
- **Metadata Creation**: Upon successful upload, a new `receipt` record is created in the PostgreSQL database. This record stores metadata about the receipt, including:

- `receipt_image_urls`: An array of URLs pointing to the stored images in S3.
- `user_email`: The email of the user who uploaded the receipt.
- `household_id`: The ID of the user's current household.
- `validation_status`: Initialized to `processing_background`.
- `created_date`, `updated_date`.
- **Storage Structure**: Images are stored in an S3 bucket with a clear hierarchy, perhaps `s3://your-bucket-name/receipts/<household_id>/<receipt_id>/<image_filename>.jpg`. This links images directly to receipts and households for easy management and access control.
- **Referencing**: The `receipt_image_urls` in the `receipt` record serve as the primary reference to retrieve and display the images.
- **Security**: S3 bucket policies or signed URLs would control access to the images, ensuring only authorized users can view their household's receipts.

D. OCR Processing Logic (AWS Textract)

- **Trigger**: OCR processing is initiated automatically once a new `receipt` record with `receipt_image_urls` is successfully created in the database and its `validation_status` is `processing_background`. This typically happens shortly after the user uploads the images.
- **OCR Method**: AWS Textract's `StartDocumentAnalysis` API is used. The `receipt_image_urls` are provided as input. Textract processes the images to extract text, forms, and tables.
- **Asynchronous Processing**: Textract analysis is an asynchronous process. Upon initiating analysis, a `JobId` is returned. A webhook or polling mechanism monitors the job status.
- **Failure Handling**:
- If Textract job fails, the `receipt.validation_status` is updated to `failed_processing`.
- The error details from Textract are logged.

- A `failed_scan_log` record is created, storing the image URLs and the encountered issues.
- The user is notified of the processing failure and prompted to provide feedback on the issues.
- **Low-Quality Scans**: Textract provides confidence scores for extracted elements. If overall confidence is low, or if key fields (total amount, date, merchant) are missing or have very low confidence, the `receipt.validation_status` might be set to `review_insights` or an internal flag is set to indicate potential low quality. This may trigger additional human review or a specific UI flow to guide the user to correct data.

E. OCR Output Normalisation

- **Initial Extraction**: After AWS Textract successfully processes the receipt images, its raw JSON output (containing blocks of text, lines, words, key-value pairs, and table data) is retrieved.
- **Key Information Extraction**:
- **Total Amount**: Search for patterns like "TOTAL", "AMOUNT DUE", "BALANCE" near numerical values. Heuristic rules and regular expressions are used to identify the final total, potentially considering currency symbols.
- **Dates**: Look for common date formats (DD/MM/YYYY, MM-DD-YY, etc.). Prioritize dates near the top or bottom of the receipt.
- **Merchants/Supermarkets**: Identify large, prominent text blocks, often at the top. Cross-reference with a known list of supermarket names.
- **Line Items**: Extract table-like structures or sequential lines that contain item names, quantities, and prices. Regular expressions are heavily used to parse individual lines into `name`, `quantity`, `unit_price`, and `total_price` components.
- **Cleaning & Deduplication**:
- Remove common OCR errors (e.g., "0" vs "O", "1" vs "l").
- Remove extraneous text like advertisements, card details, or partial words.
- Deduplicate items that might appear multiple times due to OCR quirks.

- **Structure Mapping**: The cleaned data is mapped to a predefined JSON schema for a `receipt` object:

```
{
  "supermarket": "string",
  "purchase_date": "YYYY-MM-DD",
  "total_amount": "number",
  "items": [
    {
      "name": "string",
      "quantity": "number",
      "unit_price": "number",
      "total_price": "number"
    }
  ],
  // ... other fields
}
```

- **Initial Validation**: Basic checks are performed:
- Does the sum of line item totals approximately match the extracted total amount?
- Are there a reasonable number of line items?
- Are essential fields (supermarket, date, total) present?
- **Intermediate Storage**: The initially parsed and structured data is saved back to the `receipt` record in the database. This allows for subsequent enrichment by an LLM.

F. LLM (OpenAI) Parsing Logic

- **Trigger**: After the initial OCR output normalization, the structured, but raw, receipt data (including the extracted items) is sent to an LLM for further enrichment and canonicalization. This occurs as part of the background processing flow.
- **Prompt Intent**: The LLM's primary role is to:

1. **Canonicalize Item Names**: Convert varied item descriptions (e.g., "Fresh Chicken Breast 500g", "Chicken Fillets") into a consistent, standardized `canonical_name` (e.g., "Chicken Breast").
2. **Categorize Items**: Assign each item to a predefined grocery category (e.g., "Meat & Poultry", "Vegetables & Fruits").
3. **Extract Item Details**: Determine `brand`, `pack_size_value`, `pack_size_unit`, `is_own_brand`, `offer_description`, `vat_rate`, and `vat_amount` from the item description.
4. **Derive Insights**: Generate a high-level summary and highlights for the entire receipt (e.g., "High spend on processed foods," "Good value on produce").

- **Structured Output (JSON Schema)**: The prompt explicitly requests the LLM to return a JSON object conforming to a specific schema. This schema mirrors the structure of the `Receipt` entity's `items` array and `receipt_insights` object:

```
{
  "items": [
    {
      "original_name": "string", // Original text from OCR
      "canonical_name": "string",
      "brand": "string",
      "category": "enum_from_list", // Predefined categories
      "pack_size_value": "number",
      "pack_size_unit": "enum_from_list",
      "is_own_brand": "boolean",
      "vat_rate": "number",
      "vat_amount": "number",
      "offer_description": "string"
    }
  ],
  "receipt_insights": {
    "summary": "string",
    "highlights": ["string"],
    "categories_purchased": ["string"],
    "total_items": "number",
    "estimated_healthy_items": "number",
    "estimated_processed_items": "number"
  }
```

- `}`

- **Error Handling / Invalid Responses**:
- **Schema Mismatch**: If the LLM's response is not valid JSON or does not conform to the expected schema, the system will log the error and mark the `receipt.validation_status` as `failed_processing` or `review_insights`.
- **Content Errors**: If the generated content (e.g., canonical names, categories) appears nonsensical or irrelevant (e.g., a category not in the predefined list), a retry might be attempted with an adjusted prompt, or it's flagged for manual review (`review_insights`).
- **Rate Limits/API Errors**: Standard API error handling (retries with exponential backoff, circuit breakers) is implemented. If repeated failures occur, the receipt is marked as `failed_processing`.
- **Updating Database**: The successfully parsed and enriched data from the LLM is used to update the `receipt` record in the PostgreSQL database, transitioning its `validation_status` to `review_insights`.

G. Business Rules & Categorisation

- **Primary Categorisation Source**: The primary method for item categorisation is the LLM parsing logic (as described in F), which assigns items to predefined categories based on the item's `name` and `canonical_name`.
- **Predefined Categories**: A static list of `grocery_categories` is maintained (e.g., "meat_poultry", "vegetables_fruits", "dairy_eggs", "bakery_grains"). Each category has a unique slug and display name.
- **Ingredient Mapping**: For recipe ingredients and potentially complex receipt items, an `ingredient_maps` table stores mappings between raw ingredient strings (e.g., "2 ripe avocados", "chicken breast fillets") and canonical names (e.g., "Avocado", "Chicken Breast") and their associated `category`. This helps improve consistency.
- **VAT Breakdown**: Business rules are applied to calculate and distribute VAT:
- If the receipt has a VAT breakdown, it's used directly.

- If not, the system may use an estimated `vat_rate` based on category or a default rate. VAT is distributed proportionally across items based on their `price_ex_vat`.
- **User Influence/Override**:
- **Correction Logs (`correction_logs`)**: When a user corrects an item's name or category (e.g., in the "Review Insights" interface), these corrections are logged. This data can be used to retrain or fine-tune the LLM over time.
- **Manual Mapping**: Users (or admins) can manually map `raw_ingredient_string` to `canonical_name` and `category` in the `ingredient_maps` table, directly influencing future categorisation.
- **Feedback**: Users can provide `ocr_feedback` about incorrect categorisation, which is used for system improvement.

H. Database Persistence Logic (PostgreSQL - Supabase)
The app stores data in several relational tables, maintaining integrity through foreign keys and relationships:

- **`users` table**: Stores user profiles (email, full_name, role, currency, household_id, subscription tier, usage counters). `id` is primary key, `household_id` is a foreign key to `households.id`.
- **`households` table**: Stores household details (name, admin_id, invite_code, scan limits, monthly counts). `id` is primary key, `admin_id` is a foreign key to `users.id`. Users joining a household update their `household_id`.
- **`household_invitations` table**: Stores pending invitations (household_id, invitee_email, token, status, expires_at). `household_id` is a foreign key.
- **`receipts` table**: Stores all receipt data (supermarket, date, total, items array, insights, image URLs, `user_email`, `household_id`, `validation_status`). `id` is primary key, `household_id` is a foreign key to `households.id`, `user_email` is a foreign key to `users.email`.
- **Create**: A new record is created when images are uploaded.
- **Update**: The record is updated multiple times throughout the OCR and LLM processing flow (`validation_status`, `ocr_receipt_total`, `items`, `receipt_insights`). User corrections also trigger updates.

- **`budgets` table**: Stores user-defined budgets (type, amount, period, category limits, `household_id`, `user_email`). `household_id` and `user_email` are foreign keys.
- **`credit_logs` table**: Records every credit-consuming event (user_id, household_id, event_type, credits_consumed, reference_id). `user_id` and `household_id` are foreign keys.
- **`recipes` table**: Stores recipe details (title, description, ingredients array, instructions array, servings, times, tags, allergens, image_url, source_url, `household_id`, `folder_id`). `id` is primary key, `household_id` is a foreign key to `households.id`, `folder_id` to `recipe_folders.id`.
- **Create**: When a recipe is parsed or manually added.
- **Update**: When a user edits a recipe.
- **Relationships**: `recipes` can be associated with `recipe_folders`.
- **`recipe_folders` table**: Stores user-defined folders for recipes (name, `household_id`). `household_id` is a foreign key.
- **`meal_plans` table**: Stores weekly meal plans (user_email, household_id, week_start_date, recipe_selections array). `user_email` and `household_id` are foreign keys.
- **`nutrition_facts` table**: Stores canonical nutrition data (canonical_name, calories, protein, carbs, fat, etc., household_id, user_email). `household_id` and `user_email` are foreign keys.
- **`ingredient_maps` table**: Stores mappings for ingredient canonicalization (raw_ingredient_string, canonical_name, category).
- **`ocr_feedback`, `failed_scan_logs`, `correction_logs` tables**: Store user feedback and system logs related to OCR and data quality. These link back to `receipts` via `receipt_id` and to `users` via `user_email`.
- **`aggregated_grocery_data` table**: Stores aggregated pricing data for inflation tracking (store_name, item_canonical_name, latest_price, price_observations).
- **`inflation_data` table**: Stores official inflation data (country_code, statistical_source, category_slug, period_date, index_value).
- **`grocery_categories`, `official_category_mappings`, `user_country` tables**: Configuration and lookup tables.

I. Usage Tracking & Limits

- **Tracking Mechanism**:
- A `credit_logs` table records every billable event (`ocr_scan`, `recipe_parsing`, `ai_shopping_list`, `nutrition_lookup_api`, etc.) with `user_id`, `household_id`, `event_type`, and `credits_consumed`.
- The `users` table and `households` table also maintain rolling monthly counters (e.g., `monthly_scan_count`, `parsed_recipes_this_month`) that are incremented after each relevant event.
- **Limits & Tiers**:
- The `users` table has a `tier` attribute (e.g., 'free', 'standard', 'plus') and `trial_scans_left`, `trial_recipes_parsed`.
- The `households` table has `household_scan_limit` and `household_monthly_scan_count`.
- Limits are defined per subscription tier (e.g., free tier might have 4 receipt scans per month, standard has 20).
- **Thresholds & Notifications**:
- When a user or household approaches their monthly limit (e.g., 80% used), in-app notifications and/or emails are sent to warn them.
- **When Limits are Reached**:
- **Blocking**: If a user attempts an action (e.g., uploading a receipt) that would exceed their limit, the action is blocked. An error message informs them that they've reached their limit.
- **Upgrade Prompt**: The user is prompted to upgrade their subscription or household plan to increase their limits.
- **Reset**: Monthly counters are reset at the beginning of each billing cycle, typically determined by `scan_count_reset_date` in the `households` table.

J. Analytics, Charts & Aggregations

- **Data Sources**: Analytics are primarily derived from `receipts`, `budgets`, `nutrition_facts`, `aggregated_grocery_data`, and `inflation_data` tables.
- **Aggregation Logic**:

- **Spending Charts (e.g., by Category, Supermarket)**:
- Group `receipts.items` by `category` or `receipts.supermarket`.
- Sum `total_price` for each group within specified time periods (weekly, monthly, quarterly).
- Filter by `household_id` to show relevant data.
- **Budget vs. Actual Spending**:
- Compare `budgets.amount` against aggregated spending from `receipts` for the corresponding `period_start` and `period_end`.
- Break down by `category_limits` for detailed insights.
- **Inflation Tracking**:
- Combine `aggregated_grocery_data` (for specific item price trends) and `inflation_data` (for official CPI figures).
- Calculate year-over-year or month-over-month price changes for specific `item_canonical_name` or `category_slug`.
- Compare personal spending inflation (derived from user's `receipts`) against official inflation rates.
- **Nutrition Insights**:
- Aggregate `nutrition_facts` (linked to `canonical_name`) and `receipts.items` to estimate nutritional intake from purchases.
- Calculate total calories, protein, fat, etc., for items purchased over time.
- Generate charts showing macronutrient distribution or trends in healthy/processed food purchases based on item categories.
- **Time Periods**: Aggregations support various time granularities: daily, weekly, monthly, quarterly, yearly, and custom date ranges.
- **Grouping Rules**: Data is primarily grouped by:
- `household_id`
- `category` (for items)
- `supermarket`
- `purchase_date` (for time series)
- `item_canonical_name` (for specific product trends)

- **Insight Generation**: Beyond raw numbers, AI models (`receipt_insights` in `receipts` table, `AIRecommendations` component) analyze patterns in spending and nutrition to provide actionable insights and recommendations (e.g., "You spent 20% more on snacks this month").

K. External APIs (Non-OCR)

1. **OpenAI (LLM Parsing)**:
- **Purpose**: Used for receipt item canonicalization, categorization, and general insight generation (as detailed in section F).
- **Data Fetched**: Raw OCR text and partially structured receipt data.
- **How Often**: Triggered for every new receipt upload after initial OCR. Also used for recipe parsing (`functions/parseRecipe`) and AI meal plan generation (`functions/generateAIMealPlan`).
- **Caching**: Responses for specific prompts are not explicitly cached for receipts due to the dynamic nature of input, but item canonicalizations might implicitly benefit from the `ingredient_maps` entity.
- **Failures**: Handled as described in Section F (retries, flagging for review, `failed_processing` status).
2. **Brevo (Email Marketing)**:
- **Purpose**: Sending welcome emails, trial status updates, notifications, and managing contact lists.
- **Data Fetched**: No data fetched from Brevo; data is *sent* to Brevo.
- **How Often**: Triggered on user registration (`functions/updateBrevoContact`, `functions/sendWelcomeEmail`), subscription events, or other user lifecycle events.
- **Caching**: Not applicable for sending emails.
- **Failures**: Logged internally (warn level), but generally non-critical to core app functionality (e.g., welcome email failure doesn't block user login).
3. **NutritionIX API (Nutrition Data)**:
- **Purpose**: Fetching detailed nutritional information for specific food items.

- **Data Fetched**: Calories, protein, carbs, fat, fiber, etc., per 100g or per serving.
- **How Often**: On demand when a new `canonical_name` appears in a receipt or recipe for which no `nutrition_fact` exists. Also via `functions/calorieNinjasNutrition`.
- **Caching**: Responses are cached in the `nutrition_facts` table. `failed_nutrition_lookups` tracks items that couldn't be found to avoid repeated unproductive API calls.
- **Failures**: Logged, and a `failed_nutrition_lookup` record is created to prevent repeated attempts for the same unfound item. User might be informed if a nutrition lookup specifically requested fails.

4. **ONS Data API (Inflation Data - inferred)**:

- **Purpose**: Fetching official Consumer Price Index (CPI) and inflation data for various categories and regions. (`functions/fetchONSInflationData`)
- **Data Fetched**: Time series data for index values and inflation rates.
- **How Often**: Periodically (e.g., monthly) via a scheduled job to update the `inflation_data` table.
- **Caching**: Data is persistently stored in the `inflation_data` table.
- **Failures**: Logged for administrative review. Retry mechanisms for scheduled jobs.

L. Error Handling & User Feedback

- **Validation**:
- **Client-Side**: Basic form validation prevents invalid input from reaching the backend (e.g., required fields, correct formats for dates/numbers).
- **Backend Input Validation**: All incoming API requests (payloads, query parameters) are strictly validated against expected schemas and business rules (e.g., ensuring `household_id` belongs to the authenticated user). Invalid input results in 400 Bad Request errors.
- **Error Logging**:

- All critical errors (e.g., database write failures, unhandled exceptions, external API failures) are logged internally for developers to monitor and resolve.
- Non-critical warnings (e.g., Brevo email sending issues) are logged at a lower severity.
- **User Feedback**:
- **Explicit Errors**: For user-triggered actions (e.g., "save recipe", "upload receipt"), if a process fails, a user-friendly error message is displayed in the UI (e.g., "Failed to save recipe. Please try again.").
- **Status Indicators**: Long-running background processes (like OCR) have their status tracked in the database (`receipt.validation_status`). The UI updates to reflect "Processing...", "Review Insights", or "Failed Processing".
- **Notifications**: For significant asynchronous failures (e.g., receipt scan failed catastrophically), the user receives an in-app notification.
- **Correction Prompts**: For "low-quality" or `review_insights` status, the UI guides the user to a correction interface.
- **Silent Failures**: The goal is to minimize silent failures. If an error cannot be recovered from or explicitly communicated to the user, it is at least logged for internal monitoring.
- **Security Errors**: Authentication (401 Unauthorized) and authorization (403 Forbidden) errors are explicitly returned when security policies are violated.

M. Background / Async Behaviour
Several backend operations are performed asynchronously to prevent blocking the user interface and to handle long-running tasks:

1. **Receipt Processing**:
- **Trigger**: Image upload initiates a background process.
- **Flow**: After images are uploaded and a `receipt` record is created (with `validation_status: processing_background`), a message is queued (e.g., to a message broker like AWS SQS or a Supabase/PostgreSQL function queue).
- **Worker**: A dedicated worker process picks up the message and performs:

- AWS Textract OCR.
- Initial OCR output normalization.
- LLM parsing for canonicalization, categorization, and insights.
- **Status Tracking**: The `receipt.validation_status` field (e.g., `processing_background`, `review_insights`, `failed_processing`) in the database is continuously updated by the worker to reflect the current state.
- **User Communication**: The frontend polls this status or uses real-time subscriptions to the `receipt` entity to show progress to the user (e.g., "Processing receipt...", "Ready for Review").

2. **Recipe Parsing**:

- **Trigger**: User provides a recipe URL or raw text for parsing.
- **Flow**: Similar to receipt processing, a background task extracts ingredients and instructions, potentially using an LLM, and creates a `recipe` entity.
- **Status Tracking**: The UI might show a loading spinner or a temporary status while parsing is in progress.

3. **Scheduled Jobs**:

- **Purpose**: Regularly performing maintenance, data aggregation, or external API calls.
- **Examples**:
- `functions/checkTrialStatuses`: Checks user trial periods and updates tiers.
- `functions/computeWeeklySummary`: Generates weekly spending summaries.
- `functions/fetchONSInflationData`: Periodically updates inflation data from external sources.
- `functions/enforceDataRetention`: Deletes old data according to retention policies.
- **Mechanism**: These are typically cron jobs or scheduled serverless functions that run at predefined intervals.
- **Status Tracking**: Success/failure of these jobs are logged internally.

4. **Email Sending**:

- **Trigger**: User registration, password resets, notifications.
- **Flow**: Email sending (e.g., via Brevo) is often offloaded to a non-blocking background task to ensure the primary user flow isn't delayed.

- **Status Tracking**: Failures are typically logged but don't block user experience.

Base44

Building on the previous explanations, here's how `make.com` and other features fit into your app's architecture:

A. Make.com Integration (for Email Forwarding)

- **Role of Make.com**: `Make.com` (or similar automation platforms like Zapier, IFTTT) would act as a middleman between an email service provider and your app's backend. It's used to trigger actions in your app based on incoming emails.
- **Email Forwarding Flow**:

1. **Email Receipt**: A user forwards a receipt image (or an email containing receipt details) to a dedicated email address (e.g., `scan@your-app.com`).
2. **Make.com Trigger**: Make.com has a "new email" trigger configured to monitor this dedicated inbox.
3. **Attachment/Content Extraction**: Make.com extracts any attached images or relevant text content from the email body.
4. **Backend Webhook Call**: Make.com then makes an HTTP POST request (a webhook) to a specific endpoint in your app's backend (e.g., `your-app.com/api/process-email-receipt`). The request payload would include the extracted image URLs (if uploaded by Make.com to a temporary storage) or the raw email content.
5. **App Backend Processing**: Your app's backend function (`functions/processIncomingEmail`) receives this webhook. It would then:

- Authenticate the request (e.g., via an API key shared with Make.com).
- Identify the user by the sender's email address.
- Proceed with the standard receipt processing flow (uploading images to S3, initiating OCR, LLM parsing, etc.), attributing the receipt to the user who sent the email.

- **Benefits**: This decouples email receiving from your core app logic, allows for flexible email providers, and enables non-technical users to set up custom workflows without writing code.

B. Recipe Management, Meal Planning, & AI Suggestions
These features are tightly integrated to enhance a user's food preparation and shopping experience:

- **Recipe Management**:

1. **Creation**: Users can add recipes in two main ways:

- **Manual Entry**: Inputting recipe details (title, ingredients, instructions, etc.) directly.
- **AI Parsing**: Providing a recipe URL or text, which is then sent to an LLM (as described in section F) to extract and structure ingredients, instructions, and metadata (`functions/parseRecipe`).

2. **Storage**: Each recipe is stored as a `recipe` entity, containing structured data like `title`, `ingredients` (structured array), `instructions` (structured array), `servings`, `prep_time_minutes`, `cook_time_minutes`, `tags`, `allergens`, `image_url`, `source_url`.
3. **Organization**: Users can organize recipes into `recipe_folders`.
4. **Notes & Ratings**: Users can add personal `recipe_notes_ratings` to recipes, providing feedback and tracking adjustments.

- **Meal Planning**:

1. **Creation**: Users create `meal_plan` entities for a specific week, associating `recipe` IDs with particular days, meals (breakfast, lunch, dinner, snack), and servings.
2. **AI Meal Plan Generation**: Users can request AI to generate meal plans (`functions/generateAIMealPlan`). An LLM receives user preferences (dietary restrictions, preferred cuisines, available ingredients, desired difficulty) and available `recipe` entities, and suggests a balanced meal plan, which is then stored as a `meal_plan` entity.
3. **Shopping List Generation**: From a `meal_plan`, the system aggregates all ingredients across selected recipes, scales them by serving size, and consolidates identical ingredients. This processed list forms a `shopping_list`.

- **AI Suggestions (Beyond OCR)**:

1. **Recipe Recommendations**: Based on a user's past `meal_plan` selections, `receipt` data (purchased items), and stated preferences, an LLM provides `AI_recipe_recommendations`. The prompt would include the user's history and taste profile to suggest new recipes or variations (`functions/getAIRecipeRecommendations`).
2. **Budget Insights**: AI analyzes spending patterns from `receipt` data and `budget` goals to offer personalized recommendations on where to save or optimize spending.
3. **Nutrition Analysis**: The app leverages `nutrition_facts` (from API lookups) to provide insights on meal plan health, highlighting macro-nutrient balances or suggesting adjustments.

Base44

Okay, I will clarify these points for you, ensuring platform-agnostic explanations as requested.

K. External APIs (Non-OCR) - Correction & Validation
My apologies for the previous oversight. The correct nutrition data API used is **Calorie Ninjas**, not NutritionIX.

1. **Calorie Ninjas API (Nutrition Data)**:

- **Purpose**: Fetches detailed nutritional information for specific food items.
- **Data Fetched**: Calories, protein, carbs, fat, fiber, sugar, sodium per 100g (or serving size).
- **How Often**: On demand when a new `canonical_name` appears in a receipt or recipe for which no `nutrition_fact` (sourced from Calorie Ninjas) currently exists. Also available through `functions/calorieNinjasNutrition` for explicit lookups.
- **Caching**: Responses are persistently stored in the `nutrition_facts` database table. `failed_nutrition_lookups` records track items that yielded no data from Calorie Ninjas, preventing repeated unproductive calls.
- **Failures**: Failures are logged, and a `failed_nutrition_lookup` record is created. Users might receive an in-app notification if a direct nutrition lookup fails.
- **Validation**: The `NutritionFact` and `FailedNutritionLookup` entity schemas explicitly list "calorie_ninjas" as the source, validating this correction.

Personal Insights (OpenAI Analysis)
Personal insights are a key feature where AI leverages various data points to provide actionable intelligence to the user.

1. **Receipt-Level Insights**: As previously explained, after OCR and initial parsing, an LLM (such as OpenAI's GPT models) generates `receipt_insights` for each processed `receipt`. These include a `summary`, `highlights` (e.g., "High spend on processed foods"), `categories_purchased`, and health estimations (`estimated_healthy_items`, `estimated_processed_items`). These are stored directly within the `receipt` entity.
2. **Aggregated Personal Insights**: Beyond individual receipts, the system aggregates data across multiple receipts, meal plans, and user behavior over time. An LLM analyzes these aggregated patterns to:

- **Identify Spending Trends**: "You've increased your spending on hot beverages by 15% this month."
- **Suggest Budget Adjustments**: "Consider reducing your meat purchases by 10% to stay within your protein budget."
- **Provide Nutritional Overviews**: "Your diet is consistently high in carbs; explore more balanced recipes."
- **Behavioral Nudges**: "You frequently buy brand X; have you tried our budget-friendly alternative Y?"
- **Recipe Recommendations**: The `AIRecommendations` component and the `functions/getAIRecipeRecommendations` backend function use an LLM to suggest recipes tailored to user preferences (dietary, cuisine, cooking time, excluded allergens) and past consumption patterns.

3. **Prompt & Output**: The LLM receives aggregated data (e.g., a user's last 3 months of spending, categories of items bought, recipes viewed) and a prompt requesting specific insights or recommendations, structured into a predefined JSON schema for easy consumption by the frontend.
4. **Presentation**: These insights are typically displayed on a "Personal Insights" dashboard page, offering users a high-level view and drill-down capabilities into personalized advice.

National Statistics Categories and Inflation Tracking
The app connects user spending to broader economic trends using national statistics.

1. **Internal Categories (`GroceryCategory`)**: The app maintains its own set of standardized grocery categories (e.g., "meat_poultry", "vegetables_fruits", "bakery_grains"). All receipt items are mapped to these internal categories.
2. **Official Category Mapping (`OfficialCategoryMapping`)**: To align with national economic data, these internal categories are explicitly mapped to official statistical classifications, such as COICOP (Classification of Individual Consumption by Purpose) codes used by statistical agencies like the UK's ONS (Office for National Statistics) or the US's BLS (Bureau of Labor Statistics).

- Each mapping specifies a `country_code`, `statistical_source`, the app's `category_slug`, and the `official_code` and `official_name` from the national statistics.

3. **User Country (`UserCountry`)**: Each user is associated with a `country_code` and a `statistical_source` (e.g., GB and ONS). This determines which national inflation data is relevant to them.
4. **Inflation Data (`InflationData`)**: This entity stores time-series data (e.g., monthly) for various `category_slug`s and `statistical_source`s. It records the `index_value` and, optionally, `inflation_rate_yoy` (year-over-year).
5. **Functionality**:

- **Official Inflation Tracking**: The app fetches and stores official inflation data from external APIs (e.g., `functions/fetchONSInflationData`) on a schedule. This data is normalized and stored in `InflationData`.
- **Personal Inflation Tracking**: By combining a user's `receipt` data (item prices and quantities) with their associated `UserCountry` and the `OfficialCategoryMapping`, the app can calculate a personalized inflation rate. This compares how much the user's specific basket of goods has changed in price over time against both their own past purchases and official statistics.
- **Analytics & Charts**: These official and personal inflation rates are presented visually in analytics dashboards, allowing users to see how inflation impacts their specific spending habits.

Feature Coverage
Based on our discussions, we have covered the core features and functionality of your app, including:

- User Authentication, Authorization, and Session Management.

- Household Management and Invitations.
- Receipt Upload via images or email forwarding (Make.com).
- End-to-end Receipt Processing: OCR (AWS Textract), Normalisation, and LLM Parsing (OpenAI) for item canonicalization, categorization, and receipt insights.
- Comprehensive Business Rules for data processing and categorisation.
- Database Persistence using PostgreSQL principles for all entities and their relationships.
- Usage Tracking and Limits for subscription tiers and households.
- Advanced Analytics, Charts, and Aggregations, including spending breakdowns, budget tracking, and inflation analysis (personal and official).
- External API Integrations for nutrition data (Calorie Ninjas) and national statistics (e.g., ONS).
- Error Handling, Logging, and User Feedback mechanisms.
- Asynchronous and Background Processing for long-running tasks.
- Recipe Management (manual entry, AI parsing, folders, notes/ratings).
- Meal Planning, including AI-generated plans and shopping list creation.
- AI-driven Personal Insights and Recipe Recommendations.

This comprehensive overview addresses the data flow and business logic underpinning your application's capabilities.

- Here's an explanation of your app's data categories and storage mechanisms:

1. **Receipt Spend Categories**: Receipt items are categorized into the following types:

- Hot Beverages
- Fruit
- Vegetables
- Meat & Poultry
- Fish & Seafood
- Dairy & Eggs
- Bakery & Grains
- Oils & Fats

- Sweet Treats
- Pantry Staples
- Soft Drinks
- Ready Meals
- Alcohol
- Other Food
- Toiletries
- Household Cleaning
- Pet Care
- Baby Care
- Health & Beauty
- Other Non-Food

2. **Nutritional Database and Export**:

- Your app utilizes the **Calorie Ninjas** API as its nutritional data source.
- This data is stored within your app's `NutritionFact` database entity.
- To export this data, you would programmatically access the `NutritionFact` entity using the app's SDK and then process it into your desired format (e.g., JSON or CSV). There isn't a direct built-in user interface export feature for this data.

3. **Other Databases/Entities**: Your app holds data in several other entities within its PostgreSQL database, including:

- `Users`
- `Households`
- `Receipts`
- `Budgets`
- `CreditLogs`
- `Recipes`
- `MealPlans`
- `IngredientMaps`
- `OCRFeedback`

- `FailedScanLogs`
- `CorrectionLogs`
- `AggregatedGroceryData`
- `InflationData`
- `GroceryCategories`
- `OfficialCategoryMappings`
- `UserCountry`
- `TestRuns`
- `OCRQualityLogs`
- `HouseholdInvitations`
- `RecipeFolders`

4. **Receipt Image Storage**: Receipt images are stored in **AWS S3** buckets, not directly within the app's PostgreSQL database. The database stores references (URLs) to these images.