Receipt Budgetizer & Scanner

Complete Technical Documentation

Author: Backend & Frontend Development Team
Version: 2.1 (Comprehensive)
Last Updated: December 16, 2025

Abstract

This document provides a comprehensive technical overview of the Receipt Budgetizer application. It details the system architecture, the hybrid AI/Regex parsing engine, the database schema, and the frontend implementation logic. The focus is on explaining the "why" and "how" of the system's design, with minimal code snippets provided only where necessary to illustrate complex logic.

Contents

Part I

Receipt Scanner Backend API

Chapter 1

Overview

The backend service is a specialized Flask application designed to bridge the gap between raw image data and structured financial information. It leverages Optical Character Recognition (OCR) to read text from images and employs two distinct strategies—Regex Pattern Matching and Artificial Intelligence—to interpret that text.

The system is designed to be stateless and scalable, accepting image uploads via REST API endpoints and returning JSON-formatted data that the frontend can immediately consume and display to the user.

Technology Stack

- **Flask (Python):** Chosen for its lightweight nature and rich ecosystem of data processing libraries.
- **Tesseract OCR:** An open-source OCR engine used to extract raw text from images. It serves as the foundational layer for both parsing methods.
- **Groq API (LLaMA 3):** Provides the intelligence for the AI parsing mode, capable of understanding complex receipt layouts that regex cannot handle.
- **Pillow (PIL):** Handles image pre-processing to improve OCR accuracy.

Chapter 2

Parsing Strategies

2.1 Strategy 1: Regex-Based Parsing

This is the default, cost-effective method for parsing receipts. It relies on the fact that most receipts follow a predictable structure: a store name at the top, a date, a list of items with prices, and a total at the bottom.

2.1.1 How It Works

The system applies a series of regular expressions to the raw text output from Tesseract. It looks for "anchors"—specific patterns like currency symbols, date formats (e.g., DD/MM/YYYY), and keywords like "Total" or "Tax".

Key Logic:

1. **Line Filtering:** The system iterates through every line of text, discarding noise (empty lines, short separators).
2. **Item Detection:** It looks for lines ending in a number with two decimal places, which typically indicates a price.
3. **Quantity Extraction:** If a line contains patterns like "2 x" or "3 @", the system extracts the quantity and calculates the unit price mathematically.

```python
# Captures description (group 1) and price (group 2) at the end of a line
item_pattern = re.compile(r'(.*?)\s*[\$]?\s*(\d+[.,]\d{2})$')
```

Listing 2.1: Core Regex Pattern for Items

2.2 Strategy 2: AI-Powered Parsing

For receipts with complex layouts, crumpled paper, or handwriting, regex often fails. The AI strategy sends the raw OCR text to a Large Language Model (LLM) with a specific set of instructions.

2.2.1 Prompt Engineering

The success of this method depends entirely on the "System Prompt"—the instructions given to the AI. We use a "One-Shot" prompting technique where we provide the AI with a strict JSON schema it must follow. This ensures the output is always machine-readable.

The prompt explicitly instructs the AI to:

- Fix OCR typos (e.g., correcting "Wallmart" to "Walmart").
- Infer missing quantities (defaulting to 1).
- Categorize items based on their description context.
- Return strictly valid JSON without any markdown formatting.

// Chapter 3

Core Logic Implementation

3.1 Confidence Scoring

To give users an indication of scan quality, the backend calculates a confidence score. Tesseract provides a confidence level (0-100) for every individual word it recognizes.

The system aggregates these scores by:

1. Filtering out empty blocks or noise.
2. Summing the confidence of all valid words.
3. Dividing by the total word count to get an average.

This score is crucial for the "Anomaly Detection" feature in the frontend, where low-confidence scans are flagged for manual review.

3.2 AI Categorization

Even when using the Regex method, the system can optionally use AI solely for categorization. This is a cost-saving hybrid approach. Instead of asking the AI to parse the whole receipt, the backend extracts just the item descriptions and sends a lightweight list to the AI, asking it to map each description to a fixed list of categories (Groceries, Transport, etc.).

Part II

Receipt Budgetizer Project Documentation

Chapter 4

Project Overview

Receipt Budgetizer is a comprehensive financial management tool built on the Next.js framework. Unlike simple expense trackers, it focuses on the *source* of the data—the receipt itself. By automating data entry through scanning and providing intelligent validation, it bridges the gap between physical spending and digital tracking.

4.1 Core Philosophy

The application is built around three pillars:

1. **Automation:** Minimizing manual data entry through OCR and AI.
2. **Validation:** actively checking data quality to prevent errors from entering the budget.
3. **Insight:** Transforming raw transaction data into actionable budget analytics.

Chapter 5

System Architecture

5.1 High-Level Design

The application follows a modern serverless architecture using Supabase as a Backend-as-a-Service (BaaS). This allows the frontend to communicate directly with the database and authentication services without a dedicated middleware server for standard CRUD operations.

5.1.1 Data Flow

1. **Ingestion:** The user uploads an image. The frontend sends this to the Flask OCR service.
2. **Processing:** The OCR service returns structured JSON data.
3. **Validation:** The frontend runs this data through the AI Validation Engine (client-side) to flag potential errors before saving.
4. **Persistence:** Validated data is saved to Supabase (PostgreSQL). The image is uploaded to Supabase Storage.
5. **Aggregation:** Database triggers or client-side logic updates the budget totals based on the new transaction.

5.2 Database Schema Design

The database is normalized to ensure data integrity. Instead of storing everything in one table, data is split into logical entities.

5.2.1 Key Entities

- **Receipts:** The parent entity. Stores the store name, date, total amount, and the link to the stored image. It also holds an array of "Anomalies" (flags for duplicates, spikes, etc.).
- **Receipt Items:** The child entity. Contains individual line items (milk, bread, etc.). Crucially, this table includes `ai_validation_flags` and `ai_confidence` columns to store the quality assessment for each specific item.
- **Budgets:** Links a user, a category, and a month to a specific spending limit. It caches the `spent_amount` to avoid expensive recalculations every time the dashboard loads.
- **Categories:** A master list of expense types (Groceries, Utilities) with associated icons and colors.

Chapter 6

Feature Implementation Details

6.1 AI Validation System

One of the project's most advanced features is the client-side AI Validation System. Before a receipt is saved to the database, the application analyzes the parsed data to detect logical inconsistencies. This is not just checking for empty fields; it performs heuristic analysis.

6.1.1 Validation Heuristics

The system runs five specific checks on every line item:

1. **Price Suspicious:** Checks if the price is negative, zero, or statistically improbable (e.g., a single item costing 80% of the receipt total).
2. **Quantity Unusual:** Flags non-integer quantities (unless unit is kg/lb) or zero/negative quantities.
3. **Description Unclear:** Identifies descriptions that are too short (e.g., "Misc") or contain OCR artifacts.
4. **Calculation Error:** Mathematically verifies that $Quantity \times UnitPrice \approx TotalPrice$. It allows for a small floating-point tolerance (2 cents).
5. **Category Mismatch:** Uses keyword matching to check if an item's assigned category seems contradictory to its description.

```
// Example of the calculation check logic
const expectedTotal = item.quantity * item.unitPrice;
const tolerance = 0.02;

if (Math.abs(expectedTotal - item.totalPrice) > tolerance) {
    flags.push('total_calculation_error');
    confidenceScore -= 0.25; // Reduce confidence significantly
}
```

Listing 6.1: Validation Logic Snippet

6.2 Anomaly Detection Engine

While AI Validation checks individual items, the Anomaly Detection Engine looks at the receipt as a whole and its relationship to historical data. This runs during the "Save" process.

6.2.1 Detection Rules

- **Duplicate Detection:** Queries the database for existing receipts with the same Store, Date, and Total Amount. If found, the new receipt is flagged as a potential duplicate.
- **Spending Spikes:** Calculates the user's average spending for similar receipts. If the current total is more than 3$\times$ the average, it is flagged as a "Spike".
- **OCR Mismatch:** If the raw OCR confidence score returned by the backend is below 70%, the receipt is flagged to warn the user that the text extraction might be unreliable.

Chapter 7

User Interface Experience

7.1 Dashboard

The dashboard serves as the command center. It is designed to provide immediate situational awareness.

- **Stat Widgets:** Show real-time "Remaining Budget" and "Total Spent".
- **Charts:** A pie chart visualizes spending distribution, while a line chart shows spending trends over the month.
- **Recent Activity:** A list of the latest receipts, with visual badges indicating any detected anomalies.

7.2 Receipt Detail View

This is the most complex view in the application, designed for auditing. It features a split-pane layout:

- **Left Pane:** Displays the original receipt image. Users can zoom and pan to verify details.
- **Right Pane:** Shows the extracted data.
- **Attention Section:** A dedicated, amber-colored section at the top lists any items flagged by the AI Validation System. It explains *why* an item was flagged (e.g., "Price appears suspicious") and suggests a fix.

Chapter 8

Setup Deployment

8.1 Prerequisites

To run this project, you need a standard modern web development environment: Node.js (v18+), a Supabase account, and Python (for the OCR service).

8.2 Installation Overview

1. **Database Migration:** The most critical step. You must run the SQL scripts provided in the repository to set up the tables and, importantly, the new AI validation columns. Without this, the application cannot persist validation flags.
2. **Environment Variables:** The application requires connection strings for Supabase and API keys for OpenAI/Groq. These are stored in `.env.local`.
3. **Seeding:** Helper scripts (`seed-categories.ts`) are provided to populate the database with initial categories, ensuring the app isn't empty upon first launch.

Chapter 9

Troubleshooting Guide

9.1 Common Issues

9.1.1 AI Flags Not Saving

If you see validation warnings during the scan but they disappear after saving, the database schema is likely outdated. **Solution:** Run the `add-ai-validation-columns.sql` migration script in the Supabase SQL Editor. This adds the JSONB columns required to store the flags.

9.1.2 OCR Service Errors

If the upload fails immediately, the Flask backend is likely not running or not reachable. **Solution:** Ensure the Python service is active on port 5000 and that no firewall is blocking the connection.

9.1.3 Incorrect Budget Totals

If the "Spent" amount in the budget doesn't match the receipts, the aggregation logic might be out of sync. **Solution:** Check that the receipt dates fall strictly within the month defined in the budget. The system filters strictly by the first and last day of the month.