## **Final Prompts for the ProfitFlow Project**

### **Phase 1: Project Setup & Core Structure**

**Prompt 1: Initial Flask Project Setup and Database Integration**

"Let's start by setting up your Flask project. Please generate the basic Flask application structure, including a requirements.txt file, a config.py for settings, and an app.py (or main.py) file. Integrate SQLite as the database using Flask-SQLAlchemy. Define a simple User model for authentication purposes, and set up the database initialization. Also, include a basic home route and a templates folder with a simple index.html."

### **Phase 2: User Authentication**

**Prompt 2: User Registration and Login Functionality**

"Now, let's implement user authentication. Extend the app.py to include routes for user registration and login. Use Werkzeug.security for password hashing. Create corresponding HTML templates (register.html, login.html) with forms for these actions. Ensure that users are redirected to the home page upon successful login and that their session is managed using Flask's session object. Add a logout route as well."

### **Phase 3: Data Models & Receipt Management**

**Prompt 3: Receipt and Category Data Models**

"Next, define the core data models for your application. Create a Receipt model that includes fields for user\_id (foreign key to User), amount, description, transaction\_type (income/expense), and created\_at (with a default of datetime.utcnow). Also, create a Category model for organizing transactions, linked to a user. Update your app.py and database setup to reflect these new models."

**Prompt 4: Creating and Viewing Receipts**

"Implement the functionality for users to create new receipts. Create a Flask route and an HTML form (create\_receipt.html) where authenticated users can input transaction details (amount, description, type, category). Save this data to the Receipt model in the database. Also, create a route and a template (view\_receipts.html) to display a list of all receipts for the currently logged-in user."

### **Phase 4: Gemini-Powered Voice and Image Input**

**Prompt 5: Integrate Gemini API for Audio Transaction Processing**

"Let's integrate the powerful **Google Gemini API** to process audio input and intelligently extract transaction details. This will replace the previous offline voice recognition approach.

**Your task is to:**

1. **Install the necessary Python SDK:** Include the command to install google-generativeai.
2. **Securely Load API Key:** Show how to load your Gemini API key (e.g., from an environment variable) and ensure the application correctly handles its absence.
3. **Initialize Gemini Model:** Set up the GenerativeModel instance, specifying gemini-pro.
4. **Create a New Flask Route (/process\_audio):**
   * This route should accept a POST request with an audio file (.mp3, .wav, etc.).
   * It needs to read the audio file's bytes.
5. **Craft the Gemini Prompt:** Write a detailed and clear prompt string that instructs the Gemini model to:
   * Extract amount (float), transaction\_type ('income' or 'expense'), and description (string) as **required fields**.
   * Optionally extract category (string) and transaction\_date (YYYY-MM-DD format).
   * Crucially, instruct Gemini to return these details in a **JSON format**.
   * Provide a clear example input and expected JSON output within the prompt itself to guide Gemini.
6. **Make the Gemini API Call:** Implement the Python code to send the audio bytes and your carefully crafted prompt to the Gemini API using the generate\_content method.
7. **Parse and Store Details:** Parse the JSON response received from Gemini into a Python dictionary. Then, demonstrate how to store these extracted details (e.g., amount, transaction\_type, description) into your SQLite database's Receipt model. Include a placeholder for where the actual database commit would happen.
8. **Return JSON Response:** The Flask route should return a JSON response indicating success and include the extracted details, or an error message if the processing fails.
9. **Include Error Handling:** Add basic try-except blocks to catch potential errors during API calls or JSON parsing.

This step will make your application much smarter at understanding spoken transactions!"

**Prompt 6: Integrate Gemini API for Image (OCR) Transaction Processing**

"Now, let's use the **Google Gemini API** again, specifically its multimodal capabilities, to process image inputs (like photos of receipts) and extract transaction details. This will replace any previous offline OCR approach.

**Your task is to:**

1. **Ensure google-generativeai is installed** and your GEMINI\_API\_KEY is loaded (as done for audio).
2. **Initialize Gemini Model (if not already done for vision):** If you're using gemini-pro-vision for images, ensure it's correctly initialized. If gemini-pro works directly with images for your needs, you can stick with that.
3. **Create a New Flask Route (/process\_image):**
   * This route should accept a POST request with an image file (.jpg, .png, etc.).
   * It needs to read the image file's bytes.
4. **Craft the Gemini Prompt for Images:** Write a detailed and clear prompt string that instructs the Gemini model to:
   * Examine the provided image (e.g., a receipt).
   * Extract amount (float), transaction\_type ('income' or 'expense'), and description (string) as **required fields**.
   * Optionally extract category (string), transaction\_date (YYYY-MM-DD format), and merchant (string).
   * Crucially, instruct Gemini to return these details in a **JSON format**.
   * Provide a clear example input (describing a receipt image) and expected JSON output within the prompt itself to guide Gemini.
5. **Make the Gemini API Call:** Implement the Python code to send both the image bytes and your carefully crafted prompt to the Gemini API using the generate\_content method. The content list for generate\_content will contain both the text prompt and the image data (e.g., [{'mime\_type': 'image/jpeg', 'data': image\_bytes}]).
6. **Parse and Store Details:** Parse the JSON response received from Gemini into a Python dictionary. Then, demonstrate how to store these extracted details into your SQLite database's Receipt model. Include a placeholder for where the actual database commit would happen.
7. **Return JSON Response:** The Flask route should return a JSON response indicating success and include the extracted details, or an error message if the processing fails.
8. **Include Error Handling:** Add basic try-except blocks to catch potential errors during API calls or JSON parsing.

This integration will allow users to simply snap a picture of a receipt and have the AI intelligently extract the transaction details."

### **Phase 5: UI/UX & Deployment**

**Prompt 7: Enhance User Interface and Experience (UI/UX)**

"Now that the core functionality is in place, let's focus on elevating the user experience through improved visuals, animations, and subtle sound effects.

**Your goal is to transform the aesthetic and interactivity of all your application's pages (register, login, create/view receipt, admin dashboard, etc.). Specifically, generate instructions and code snippets for:**

1. **Modernizing the UI/UX:**
   * **Frontend Framework/Library:** Recommend and integrate a lightweight frontend framework or library (e.g., **Bootstrap**, **Tailwind CSS**, or a simple CSS framework like **Milligram** or **Pure.css**) to provide a clean, modern, and responsive design across all pages. Focus on usability and readability for small screens (mobile-first approach).
   * **Color Palette and Typography:** Suggest a cohesive and professional color palette that aligns with a financial tracking app (e.g., shades of blue, green, or grey for trust and clarity). Recommend legible fonts.
   * **Iconography:** Integrate a free icon library (e.g., **Font Awesome**, **Material Icons**) to visually enhance buttons, navigation, and data points.
2. **Adding Engaging Animations:**
   * **Micro-interactions:** Implement subtle animations for common user interactions. Think about:
     + **Button hovers and clicks:** Small scaling or color changes.
     + **Form input focus:** A gentle border highlight or shadow.
     + **Page transitions:** A simple fade-in or slide-in effect when navigating between routes (this might involve a bit of JavaScript).
     + **Feedback animations:** A quick checkmark or a small shake on successful/failed form submissions.
   * **Loading indicators:** A custom loading spinner or animation when processing Gemini API requests, ensuring the user knows something is happening.
3. **Integrating Subtle Sound Effects:**
   * **User feedback:** Suggest and implement very subtle, non-intrusive sound effects for key actions:
     + **Successful submission:** A gentle "ding" or "chime" when a receipt is successfully saved.
     + **Error/Validation warning:** A soft "click" or "thud" for form validation errors.
     + **Voice/Photo capture:** A quick "shutter" sound for photo input or a "record start/stop" sound for voice.
   * **Implementation:** Explain how to embed these sounds (e.g., using HTML5 <audio> tags triggered by JavaScript) and emphasize keeping them low volume and optional.

This ambitious step will make your **ProfitFlow** or **LedgerVoice** app not just functional, but genuinely enjoyable to use, fostering better adoption and retention for small traders."

**Prompt 8: Prepare for Production Deployment**

"To prepare your Flask application for production, we need to make a few adjustments. Generate instructions and code snippets for:

1. Using gunicorn as a production-ready WSGI server.
2. Setting up environment variables for sensitive data (e.g., SECRET\_KEY, GEMINI\_API\_KEY).
3. A basic Procfile if deploying to platforms like Heroku.
4. Mention the importance of setting DEBUG = False in config.py for production."