🧾 Product Requirements Document (PRD) – PropertyIQ Prototype

Prototype: https://v0-property-iq-dashboard.vercel.app/

Backend Repo: https://github.com/ansh10/propertyiq-backend

1. Problem Statement

Managing property tax documents is a repetitive, error-prone process for property owners, accountants, and municipal organizations.
These documents often exist only in **PDF or scanned formats**, containing essential information such as ownership details, assessed value, tax year, and due amounts. Extracting this data manually is time-consuming and inefficient — particularly when dealing with large volumes of properties or inconsistent document layouts.

The lack of automation in property tax data extraction leads to:

- **Delays in financial reconciliation** and reporting.
- **Human errors** during manual data entry.
- **Inefficient workflows** when reviewing multiple property files.

2. Our Solution

PropertyIQ is an **AI-powered property tax document processor** that automates the extraction of key fields from property tax PDFs.
The system enables users to:

1. **Upload** property tax PDF documents via a web dashboard (built with React and hosted on Vercel).
2. **Extract** structured data — such as owner name, property address, tax year, amount due, and due date — using OCR (Optical Character Recognition) and Python's text-processing libraries (`pytesseract`, `pdf2image`, `re`).
3. **Review and export** the extracted data in a clean, standardized format.

This prototype demonstrates a **proof of concept** that validates automated property tax extraction as a viable, scalable process that can integrate into future municipal or real estate software ecosystems.

3. Implementation Summary

- **Frontend:** React (Vercel) with drag-and-drop upload UI and AI-themed interface.
- **Backend:** Flask API (Render) for processing uploaded PDFs and returning structured JSON data.
- **Core Logic:** Text extracted from images using `pytesseract`, parsed through regular expressions for specific field patterns.

4. Limitations of the Prototype

a. Backend Performance (Render Hosting)

- Hosted on Render's free tier, resulting in:
    - **Limited CPU and memory** for OCR tasks.
    - **Timeouts** for larger or scanned documents requiring longer processing.
    - **Cold starts** after inactivity, causing 30–60 s initial delays.
- Consequently, the extraction pipeline could not be fully optimized for accuracy or performance. Hosting on a stronger infrastructure (AWS EC2, Cloud Run, or Render Pro) would yield faster, more consistent results.

b. OCR and Parsing Accuracy

- `pytesseract` performs best on clear, machine-generated PDFs but struggles with **noisy scans, rotated pages, or handwritten text**.
- Current field extraction uses **static regex rules**, limiting adaptability to varying layouts.
- No **semantic understanding or NLP-based parsing** is yet implemented.

c. No Persistent Storage

- The system processes uploads in-memory without storing results.
- There is no integration with databases (e.g., PostgreSQL, Firebase) or cloud storage, preventing data reuse or analytics.

d. Debugging and Logging

- Render's transient logs limit visibility into OCR or extraction errors.
- No persistent or centralized monitoring implemented.

e. CORS and Security Constraints

- Cross-origin requests between Vercel (frontend) and Render (backend) required explicit CORS configuration.

- File validation is basic; advanced scanning and sanitization should be added for production.

5. Future Enhancements

1. Deploy backend to a **persistent, high-performance environment**.
2. Integrate **database and authentication** for user-specific document storage.
3. Implement **async task management** (Celery + Redis) for large files.
4. Enhance extraction using **NLP models** (spaCy, LayoutLM).
5. Add **error tracking, analytics, and retry mechanisms**.