Tech Stack:

Backend:

- \*\*Python 3.11+\*\* - Core programming language

- \*\*FastAPI\*\* - Modern, fast web framework

- \*\*PostgreSQL\*\* - Primary database

- \*\*Redis\*\* - Caching and session management

- \*\*OpenCV\*\* - Image processing

- \*\*PyTorch\*\* - Machine learning models

- \*\*PaddleOCR\*\* - Text extraction from drawings

1. Python 3.11+

The core programming language that orchestrates the entire backend logic, including handling requests, processing data, interfacing with the database, and managing machine learning operations.

2. FastAPI

A high-performance web framework used to build RESTful APIs. It handles incoming HTTP requests from frontend clients or other services, routes them to appropriate functions, and returns responses. FastAPI makes it easy to define endpoints for functionalities like cost estimation, image uploads, or report retrieval.

3. PostgreSQL

The relational database system storing structured data such as project details, user information, estimation results, version histories, and other metadata. FastAPI interacts with PostgreSQL to read/write data as needed during operations.

4. Redis

An in-memory data structure store used for caching frequently accessed data or computational results, which enhances performance. It also manages session data for authenticated users, enabling quick retrieval of user sessions and temporary data during estimation processes.

5. OpenCV

A computer vision library used for processing images—specifically construction drawings or related visual inputs. It can perform tasks like image enhancement, feature detection, or extracting relevant sections of drawings, preparing data for further analysis.

6. PaddleOCR

An OCR (Optical Character Recognition) tool used to extract text from images of drawings, plans, or documents. It enables automated reading of annotations, labels, or measurements embedded within construction images or drawings, converting them into structured data.

7. PyTorch

A machine learning framework used to develop, train, and run ML models. It powers custom models that may perform tasks like estimating costs based on extracted features, predicting carbon footprint metrics, or classifying drawing components.

How They Interact in the Platform

Data Input & Processing:

Users upload images or drawings via the frontend; FastAPI receives these uploads. For processing, images are sent to OpenCV for pre-processing (e.g., noise reduction, segmentation).

Text Extraction:

Processed images are passed to PaddleOCR to read embedded text such as annotations, measurements, or labels.

Image Analysis & Machine Learning:

Extracted data, along with images, might be further processed by PyTorch models trained for specific tasks (e.g., estimating quantities or carbon footprint). These models interpret visual features and numerical data to generate estimates.

Data Storage & Retrieval:

Results, project details, and user data are stored in PostgreSQL. FastAPI handles database interactions for CRUD operations and results retrieval.

Performance Optimization:

Frequently accessed data or computationally expensive results are cached in Redis, reducing load times and improving responsiveness.

Workflow Integration:

The entire flow—from image upload, processing, text extraction, ML inference, to storing and returning results—is managed through FastAPI, coordinating the interactions among these components seamlessly.

A black screen with white text

AI-generated content may be incorrect.

Front End Tech Stack:

**1. Next.js 14**

* **What it is:** A full-stack React framework built for production, with file-based routing, SSR/SSG capabilities, and API routes.
* **Role:** Acts as the backbone of the frontend. It defines the structure of pages (UI views), manages routing (URL → page), and handles server-side or client-side rendering depending on the page requirements.
* **How it interacts:**
  + Defines **pages** where users upload drawings, review estimates, or view carbon footprints.
  + Integrates React Query for **data fetching** (e.g., calls FastAPI endpoints for estimates, drawing uploads).
  + Uses **API routes or SSR** to pre-fetch and hydrate data for better performance or SEO.

**2. TypeScript**

* **What it is:** A statically typed superset of JavaScript.
* **Role:** Adds type safety to React components, form values, API responses, and model interfaces.
* **How it interacts:**
  + Ensures **strong typing** across frontend logic, reducing runtime errors and improving developer productivity.
  + Works with Next.js and React components to define **props**, **API data models**, and **form schemas**.

**3. Tailwind CSS**

* **What it is:** A utility-first CSS framework for rapid UI development.
* **Role:** Handles styling of components and pages.
* **How it interacts:**
  + Used within JSX/TSX files alongside React to define **responsive, utility-based layouts and themes**.
  + Allows consistent styling across form inputs, buttons, dashboards, and result summaries with minimal custom CSS.

**4. React Hook Form**

* **What it is:** A performant, flexible library for managing form state and validation in React.
* **Role:** Handles user inputs—drawing uploads, form-based project metadata, manual overrides, etc.
* **How it interacts:**
  + Registers form fields, tracks state (dirty, touched, valid), and submits structured payloads to the backend.
  + Integrates with TypeScript for **typed form schemas**.
  + Often pairs with validation libraries like **Zod** or **Yup** for schema-based validation.

**5. React Query (now called TanStack Query)**

* **What it is:** A library for fetching, caching, and synchronizing server data in React.
* **Role:** Manages asynchronous communication between frontend and backend (e.g., fetching estimates, uploading files).
* **How it interacts:**
  + Defines **queries and mutations** for calling FastAPI endpoints (e.g., getEstimate, uploadDrawing, getProject).
  + Automatically **caches** results and **invalidates** stale data after updates.
  + Handles loading/error states, enabling smooth UX without manual boilerplate.

**🔄 How They All Work Together in Your Platform**

Let’s walk through a **typical user workflow** and explain how each frontend component interacts:

**🔹 1. User Uploads a Drawing**

* **Page (Next.js):** /upload
* **Form (React Hook Form):** Collects the drawing file and metadata.
* **Validation:** Ensures correct format and required fields using Hook Form + Zod/Yup.
* **Submit Handler:** Uses **React Query mutation** to call a POST /upload FastAPI endpoint.
* **TypeScript:** Ensures payloads and responses are correctly typed.

**🔹 2. Platform Processes and Returns Cost/Carbon Estimate**

* **Loading State:** React Query shows a spinner while waiting for inference.
* **Results Page (Next.js):** /project/[id]/summary
* **React Query fetches:** GET /project/:id/results (includes quantities, costs, carbon footprint).
* **Tailwind CSS:** Styles cards, tables, graphs, and KPIs in a responsive layout.
* **TypeScript:** Provides type safety for results (e.g., Estimate, FootprintMetrics).

**🔹 3. User Reviews and Edits Details**

* **Editable form (React Hook Form):** Allows user to modify extracted quantities, override assumptions.
* **On Save:** Hook Form + React Query mutation sends a PUT request to update the backend.
* **Optimistic UI:** React Query updates the UI before server confirms, improving UX.

**🔹 4. User Navigates Across Pages**

* **Next.js router:** Manages navigation between /dashboard, /upload, /project/[id]
* **Data fetching:** Uses **React Query prefetch** or SSR to preload key data
* **Tailwind:** Ensures transitions and layouts are fluid and consistent

**🔹 5. Performance Enhancements**

* React Query reduces redundant API calls by caching results.
* Redis on the backend complements this by caching processed inferences.
* SSR in Next.js 14 can be used for SEO-friendly rendering of public-facing pages or reports.