**BoQ-AI: Project Status & Technical Report**

**Date:** June 11, 2025 **Status:** Phase 2 Complete. Application is fully functional with on-screen BoQ generation, vendor pricing selection, and costed Excel export.

**1. High-Level Project Goal**

The primary objective of the BoQ-AI project is to automate the creation of a Bill of Quantities (BoQ) by directly processing CAD drawing files (.dwg or .dxf). The system extracts geometric data, applies a set of measurement rules, and generates a detailed BoQ with cost estimations, drastically reducing manual effort and improving accuracy.

**2. Current Project Status: Phase 2 Complete**

We have successfully implemented a robust full-stack application that achieves and surpasses our initial goals.

* **Core Functionality:** The application can process an uploaded CAD file and generate an accurate Bill of Quantities.
* **Interactive UI:** The results are displayed instantly in a clean, professional table within the web interface.
* **Dynamic Costing:** Users can select material vendors and prices for applicable line items, and the total project cost updates in real-time.
* **Costed Export:** The final BoQ, complete with selected vendor pricing and total cost, can be downloaded as a formatted Excel spreadsheet.
* **Stability:** All major architectural and logical bugs have been resolved. The platform is stable and ready for the next phase of development.

**3. Final File Architecture**

The project is logically separated into a backend (the processing engine) and a frontend (the user interface).

boq-ai-project/

├── backend/

│ ├── app/

│ │ ├── \_\_init\_\_.py # Initializes the 'app' directory as a Python package.

│ │ ├── main.py # The main Flask application file with API endpoints.

│ │ ├── rules.py # The "brains" of the BoQ generation; calculates quantities.

│ │ ├── excel\_writer.py # Handles the creation of the final .xlsx file.

│ │ ├── convert\_dwg\_to\_dxf.py # Utility to handle .dwg file conversion.

│ │ └── downloads/ # (Auto-generated) Temp storage for files (can be cleared).

│ └── venv/ # Python virtual environment.

│

├── frontend/

│ ├── public/

│ │ └── data/

│ │ └── vendors.json # NEW: Data file for vendor materials and pricing.

│ ├── src/

│ │ ├── services/

│ │ │ └── api.js # Handles all communication with the backend API.

│ │ ├── App.jsx # The main React component for the user interface.

│ │ ├── main.jsx # The entry point for the React application.

│ │ └── index.css # Tailwind CSS base styles.

│ ├── index.html # The main HTML file for the single-page application.

│ ├── package.json # Defines project dependencies and scripts.

│ └── vite.config.js # Configuration for the Vite build tool.

│

└── rules/

├── american\_standard.json # Rule set for American measurement standards.

└── indian\_standard.json # Rule set for Indian measurement standards.

**4. Technical Deep-Dive**

**How We Identify Elements & Dimensions**

The core of the "AI" in this project is not a machine learning model, but a sophisticated **rule-based expert system**. We treat the CAD drawing as a structured data source and apply a set of pre-defined rules to it.

1. **File Ingestion (main.py):** When a user uploads a file, the Flask backend receives it. If it's a .dwg file, it's first converted to the more open .dxf format using a helper script.
2. **Data Extraction (ezdxf library):** The backend uses the powerful ezdxf Python library to parse the .dxf file. It iterates through every entity (lines, polylines, hatches, circles, etc.) in the drawing's "modelspace."
3. **Layer-Based Grouping (\_collect\_entities\_by\_layer):** The single most important piece of information we use for identification is the **layer name**. The backend groups all entities into a Python dictionary where the keys are the layer names (e.g., A-WALL-INTR, CONCRETE\_SLAB) and the values are lists of all the geometric shapes on that layer.

**How the Rule System Works (rules.py)**

The logic for what to measure and how to describe it is entirely externalized into our .json rule files. This is a powerful design choice that allows us to add or change rules without ever touching the Python code.

1. **Rule Loading:** Based on the user's selection ("American SMM" or "Indian IS1200"), the backend loads the corresponding JSON file (e.g., american\_standard.json).
2. **Rule Structure:** Each rule in the JSON file tells the system what to do. For example:
3. "interior\_partition\_walls": {
4. "target\_layers": ["160 R.C.WALL", "160 R.C. WALL HAT", "A-WALL-INTR"],
5. "measurement\_rule": "length",
6. "boq\_description": "Interior Partition Wall, Stud and Drywall Assembly",
7. "boq\_unit": "LFT"
8. }
   * target\_layers: An array of layer names that correspond to this BoQ item. Our system will find all entities on any of these layers.
   * measurement\_rule: Tells the calculation engine which function to use (length, area, or count).
   * boq\_description & boq\_unit: The final text that will appear in the BoQ.
9. **Applying Rules (evaluate\_rules):** The system iterates through each rule. For each rule, it collects all the entities from the layers specified in target\_layers. It then passes this collection of entities to the appropriate calculation function based on the measurement\_rule.

**How Calculations and Conversions Work**

* **Geometric Calculation:** The rules.py file contains functions (get\_total\_length, get\_total\_area, count\_entities) that perform the actual measurements. We've made these robust to handle various entity types. For example, if a LWPOLYLINE doesn't have a pre-calculated length, our code manually calculates it by summing the distance between its vertices.
* **Unit Conversion:** A critical feature we implemented is automatic unit conversion. The calculation functions assume the standard drawing unit is millimeters (a common industry practice) and contain hardcoded conversion factors (e.g., MM\_TO\_FEET = 1 / 304.8) to convert the raw data into the desired output unit (e.g., LFT, SQFT).

**How Vendor & Costing Works**

* **Vendor Data (vendors.json):** The vendor pricing is completely decoupled from the backend. It lives in a simple JSON file in the frontend's public/data directory. This file maps BoQ item descriptions to a material type (e.g., "concrete") and lists available vendors and prices for that material.
* **Dynamic Frontend Calculation:** When the frontend displays the BoQ table, it checks the vendors.json file. If a BoQ item's description has a matching material type, it dynamically creates a dropdown menu for vendor selection.
* **Live Costing:** Using React's state management, every time a user selects a vendor from a dropdown, the component updates the line cost (quantity \* selected\_price) and recalculates the total cost for the entire project in real-time.

**How the Final Excel File is Generated**

The application uses a clever two-step API process to include the final costs:

1. **Step 1 (/api/boq):** The frontend sends the CAD file. The backend does all the geometric calculations and sends back **only the raw BoQ data** (description, quantity, unit) as JSON.
2. **Step 2 (/api/generate\_excel):** After the user selects vendors in the UI, they click the "Download Excel" button. The frontend sends the **complete BoQ data, now including the chosen unit price for each item**, to this new endpoint.
3. **Final Creation (write\_boq\_with\_costing):** This backend endpoint receives the final data, adds columns for "Unit Price" and "Line Cost," calculates the total, formats everything beautifully using openpyxl, and sends the final .xlsx file back to the user.

**5. Next Steps: Phase 3**

The project is in an excellent state. We have successfully separated the data extraction logic (backend) from the data presentation and interaction logic (frontend). The next logical phase is to build features on top of this stable foundation, starting with **User Authentication** to enable project saving and customization.