G1 PO App - Consolidated Handover Report

Date: May 9, 2025

Prepared For: Incoming Developer

Prepared By: Mark (via collaboration with AI Assistant Gemini)

Introduction

This document provides a comprehensive overview and handover for the "G1 PO App" project. It consolidates the initial project outline and handover report (dated May 8, 2025) with all subsequent development progress, features implemented, and bug fixes addressed up to May 9, 2025. The goal is to provide a clear and detailed guide for the incoming developer to understand the project's current state, architecture, codebase, and remaining tasks to achieve the Minimum Viable Product (MVP) and beyond.

1. Project Overview

(Based on original Handover Report, Section 1)

Goal: To develop a web application ("G1 PO App") that automates the purchase order (PO) and shipment process for drop-shipped items originating from BigCommerce orders.

Core Functionality:

* Ingest relevant orders from BigCommerce (filtered by status, e.g., "Awaiting Fulfillment").

* Provide a web interface for users to review these orders.

* Allow users to trigger processing for selected orders.

* Generate Purchase Orders (PDFs) using specific HPE Option PNs mapped from original BigCommerce SKUs. Includes logic to handle SKUs with underscores (e.g.,

`PART_OPTIONPN`) by attempting a secondary lookup on the `OPTIONPN` part if the full SKU doesn't map directly.

* Generate Packing Slips (PDFs).

* Generate UPS Shipping Labels (PDFs) via UPS API, including OAuth 2.0 token management.

* Email the generated PO, Packing Slip, and Label to the selected supplier via Postmark API.

* Upload generated documents (PO, Packing Slip, Label) to Google Cloud Storage (GCS).

* Update the order status and add tracking information in BigCommerce via its API.

* Update the local application database status for processed orders (e.g., to "Processed", "RFQ Sent").

* Provide a web interface (React/Vite) for:

   * Viewing and filtering orders (by status: New, RFQ Sent, Processed, etc.).

   * Viewing detailed order information, including customer comments from BigCommerce.

   * Interactively preparing PO details (selecting supplier, editing PO notes, adjusting purchase SKUs, descriptions, quantities, and unit costs for line items).

   * Triggering the order processing workflow.

   * (Future) Managing suppliers and product mappings.

2. Current Status (As of May 9, 2025)

(Consolidates original report Section 2 with recent developments)

Backend (Python/Flask - `app.py`):

* **Order Ingestion (`/ingest_orders`):**

   * Fetches orders from BigCommerce based on a configurable status ID (e.g., "Awaiting Fulfillment").

   * Correctly determines if an order is international based on shipping address.

  * Assigns an internal status of 'new' or 'international_manual'.

  * Saves BigCommerce `customer_message` to the local `orders.customer_notes` field.

  * Inserts new orders and their line items into the local PostgreSQL database.

  * Updates existing orders if their status or international flag changes.

  * Error handling for this endpoint has been improved.

* ****Order Retrieval (`/api/orders`, `/api/orders/<id>`):****

  * `GET /api/orders`: Fetches a list of orders, supports filtering by status (e.g., `?status=new`), and sorts by date. Returns `customer_notes`.

  * `GET /api/orders/<id>`:

    * Fetches detailed information for a single order, including its line items and `customer_notes`.

    * ****HPE Part Number & Custom Description Mapping:****

      * Implements robust logic to determine the `hpe_option_pn` for each line item:

        1. Directly maps `order_line_items.sku` to `hpe_part_mappings.option_pn`.

        2. ****Fallback Logic:**** If no direct map and `order_line_items.sku` contains an underscore (e.g., `ACTUALSKU_OPTIONPN`), it attempts to map the part **after** the last underscore.

      * Fetches the corresponding `hpe_po_description` from `hpe_description_mappings` based on the determined `hpe_option_pn`.

    * Returns `original_sku`, `line_item_name`, `hpe_option_pn`, `hpe_pn_type`, and `hpe_po_description` for each line item, allowing the frontend to display comprehensive information.

* ****Order Processing (`POST /api/orders/<id>/process`):****

  * This core endpoint is largely functional and orchestrates the multi-step drop-shipment workflow within a single database transaction.

  * ****Key Steps:****

    1. Validates input (supplier, weight, line item costs/quantities).

    2. Fetches order and supplier details.

3. Checks if the order is domestic (international orders are currently blocked from this auto-processing).

4. Generates a sequential PO number.

5. Inserts a record into the `purchase_orders` table.

6. For each line item submitted from the frontend:

    * Determines the `sku_for_po` using the same robust HPE mapping logic (direct then underscore fallback) as `get_order_details`.

    * Uses the `description` field *as submitted by the frontend* (which would have defaulted based on HPE custom description or original name, and could be manually edited) for the `po_line_items.description` and for the PO PDF.

    * Inserts records into `po_line_items` with the correct `sku_for_po` and submitted/finalized `description`, quantity, and unit cost.

7. Generates PO PDF (via `document_generator.py`) using the finalized SKUs and descriptions.

8. Generates Packing Slip PDF (via `document_generator.py`).

9. Generates UPS Shipping Label & Tracking Number (via `shipping_service.py`, handles OAuth).

10. Inserts a record into the `shipments` table.

11. Uploads all three PDFs to Google Cloud Storage (GCS) under `processed_orders/order_<id>/...`.

12. Updates `purchase_orders` and `shipments` tables with GCS paths.

13. Sends an email with all three PDF attachments to the selected supplier via Postmark (using `email_service.py`).

14. Updates the BigCommerce order by creating a shipment with the tracking number (implicitly updating its status to "Shipped" or a configured shipped status ID).

15. Updates the local `orders` table status to 'Processed'.

16. Commits the database transaction on success; rolls back on any failure.

* **Status Update (`POST /api/orders/<id>/status`):**

  * Allows specific updates to an order's status (e.g., from 'new' to 'RFQ Sent'). Used by the frontend when Brokerbin links are clicked.

* **SKU/Description Lookup (`GET /api/lookup/description/<sku>`):**

  * Provides a dynamic lookup for descriptions based on a SKU.

  * Enhanced to use the `get_hpe_mapping_with_fallback` logic:

    1. Tries input SKU against `hpe_description_mappings` (as if input is an OptionPN).

    2. If not found, tries input SKU against `products.standard_description`.

    3. If still not found, maps input SKU via `hpe_part_mappings` (with fallback) to get an OptionPN, then tries that OptionPN against `hpe_description_mappings`.

* **Supplier and Product Mapping Endpoints:** Basic CRUD endpoints for suppliers (`/api/suppliers`) and product mappings (`/api/products` - referring to the `products` table for standard descriptions, distinct from `hpe_part_mappings`) exist (details in original report).

* **Database Connection:** Cloud SQL (PostgreSQL) connection via Cloud SQL Auth Proxy (locally) is established and working.

* **Error Handling:** Basic error handling and transaction rollbacks are in place. More detailed error responses (JSON) are provided by some endpoints.

Frontend (React/Vite - `Dashboard.jsx`, `OrderDetail.jsx`):

* **Project Structure:** Basic project structure (`main.jsx`, `App.jsx`) and routing (`react-router-dom`) are in place.

* **`Dashboard.jsx`:**

  * Fetches and displays a list of orders from `/api/orders`.

  * Implements status filtering (dropdown, defaults to 'new', includes 'All').

  * **UI Enhancements:**

    * Page title "Order Dashboard" is now centered, all caps, and bold.

    * **Mobile Card View:**

      * Displays Order #, Status badge, Customer, Ship Method, Ship To, Total.

    * **Order Date & Time:** Displayed in the top-right corner of each card (date on one line, time below). The original "Order Date" label/field in the main card flow is hidden.

    * **Customer Comments:** Conditionally displays `customer_notes` if present.

    * The "Status:" text label that was overlapping the status badge has been removed.

  * Desktop table view remains, now includes a "Comments" column (shows snippet, full on hover).

  * Order rows/cards are clickable, navigating to the order detail page.

  * Hydration error related to `<div>` inside `<tr>` has been fixed by moving the date/time element into the first `<td>`.

* **`OrderDetail.jsx`:**

  * Fetches and displays comprehensive details for a single order from `/api/orders/<id>`.

  * **Order Information Section:**

    * Displays original line items, showing:

      * Original BigCommerce SKU (as a clickable Brokerbin link).

      * Mapped HPE Option PN (if different from original SKU, also a clickable Brokerbin link). This mapping uses the direct then underscore-fallback logic from the backend.

    * Displays `customer_notes` (BigCommerce order comments) if present.

  * **Interactive Processing Form ("Items to Purchase" & other sections):**

    * **Supplier Dropdown:** Populates from `/api/suppliers`; auto-fills "PO Notes" from `suppliers.defaultPOnotes` on selection.

    * **Editable PO Notes:** Textarea for custom notes.

    * **Editable "Items to Purchase" List:**

      * Defaults "Purchase SKU" to `hpe_option_pn` (from fallback logic) or `original_sku`.

      * Defaults "Description" to `hpe_po_description` (if available for the `hpe_option_pn`) or the original `line_item_name`.

      * Allows editing of "Purchase SKU", "Description" (as textarea), "Quantity", and "Unit Cost".

* "Unit Cost" is a required field for form submission.

* "Description" field dynamically updates (debounced) via `/api/lookup/description/<sku>` when "Purchase SKU" is changed.

* **Shipment Method Dropdown:** Defaults based on customer's method (maps "Free Shipping" to "UPS Ground").

* **Shipment Weight Input.**

* **"RFQ Sent" Logic:**

* Clicking on Brokerbin links (for original SKU or mapped Option PN) in the "Order Information" section will:

1. Check if the current order status is 'new'.
2. If 'new', it calls `POST /api/orders/<id>/status` to update the status to "RFQ Sent".
3. Updates the local component state to reflect the new "RFQ Sent" status.
4. Displays a confirmation message.

* **Form Disabling Logic:**

* **Input Fields** (supplier, notes, item details, shipment info): Disabled only if order status is "Processed". They remain editable if status is "RFQ Sent".

* **"PROCESS ORDER" Button:** Disabled if order status is "Processed" OR "RFQ Sent", or if an async operation (`processing`) is in progress.

* **Form Submission (`handleProcessOrder`):**

* Performs client-side validation (supplier, weight, item SKU/description/qty/cost).

* Sends a `POST` request to `/api/orders/<id>/process` with the finalized PO details.

* Displays "Processing..." text on the button during the API call.

* Handles success/error feedback messages, now displayed *above* the "PROCESS ORDER" button.

* **UI/UX:**

* "Back to Dashboard" link at the top of the page has been removed (global nav is sufficient).

    * Brief flashing of "Order details not found" message during initial load has been mitigated.

* **CSS Styling:**

  * `OrderDetail.css` and `Dashboard.css` have been updated to support the UI changes.

  * Global CSS variables are defined in `OrderDetail.css` (consider moving to `App.css` or `index.css`).

* **Placeholder Components:** `Suppliers.jsx`, `Products.jsx` exist (as per original report).

Database (PostgreSQL on Cloud SQL):

* Core table schemas (orders, order_line_items, suppliers, purchase_orders, po_line_items, shipments, products) are defined (as per original report).

* `hpe_part_mappings` table: Exists and populated (approx. 8000 rows).

* `hpe_description_mappings` table: **This feature is now complete.** This table is assumed to be created and populated with OptionPN-to-custom PO descriptions. The backend and frontend now fully utilize this for defaulting descriptions.

* `suppliers` table: Includes `defaultPOnotes` column.

* `orders` table: Includes `customer_notes` column to store BigCommerce customer messages.

Integrations:

* **UPS API:** OAuth and Label Generation (Shipment request) working in CIE test environment.

* **BigCommerce API:** Order fetching (V2 list and V2 by ID, including sub-resources like shipping addresses and products), and Shipment creation/status update (V2) are working.

* **Postmark API:** Email sending with PDF attachments is working (tested to internal domain, requires approval for external).

* **Google Cloud Storage (GCS):** Document (PO, Packing Slip, Label) upload is working.

3. Technology Stack

(Based on original Handover Report, Section 3 - Assumed Unchanged)

* **Backend:** Python 3.x, Flask, SQLAlchemy (Core API with `text()`), pg8000 (PostgreSQL driver)

* **Frontend:** React, Vite, react-router-dom

* **Database:** PostgreSQL (hosted on Google Cloud SQL)

* **Cloud:** Google Cloud Platform (GCP) - Cloud SQL, Cloud Storage (GCS), Secret Manager (Planned), Cloud Run (Planned)

* **APIs:**

  * UPS Shipping API (JSON REST, v2409 tested in CIE env)

  * BigCommerce API (V2 REST)

  * Postmark API (Email)

* **Document Generation:** ReportLab

* **Environment:** Python `venv`, `.env` file for configuration, `requirements.txt` for Python dependencies, `package.json` for Node dependencies.

4. Architecture Overview

(Based on original Handover Report, Section 4 - Assumed Unchanged)

* **Backend (Flask):** Serves a RESTful API for the frontend. Handles business logic, database interactions, and external API calls. Orchestrates the multi-step order processing workflow.

* **Frontend (React/Vite):** Single Page Application (SPA) providing the user interface for viewing orders, triggering processing, and (future) managing related data. Interacts with the Flask backend API.

* **Database (Cloud SQL):** PostgreSQL instance storing application state (orders, POs, shipments, mappings, etc.).

* **Cloud Storage (GCS):** Stores generated PDF documents (PO, Packing Slip, Label).

* **Deployment (Planned):** Containerized Flask backend and potentially the frontend build deployed to Cloud Run, using Secret Manager for credentials and connecting to Cloud SQL.

5. Codebase Structure (Key Files & Recent Changes)

(Based on original Handover Report, Section 5, with updates)

Backend (Root Directory - Python):

* `app.py`: Main Flask application file. Initializes DB engine, GCS client. Defines all API routes. Contains primary orchestration logic for processing, ingestion, and lookups.

  * **Recent Changes:**

    * Added `get_hpe_mapping_with_fallback()` helper function.

    * Modified `get_order_details()` to use the new SKU mapping helper and fetch custom descriptions.

    * Modified `process_order()` to use the new SKU mapping helper for `sku_for_po` and correctly use frontend-submitted descriptions for DB/PDF.

    * Modified `/ingest_orders` to correctly save `customer_message` to `customer_notes` and resolved bugs related to updating existing orders.

    * Enhanced `/api/lookup/description/<sku>` to use fallback mapping.

* `shipping_service.py`: Contains functions for UPS OAuth, UPS label generation, and BigCommerce order updates.

* `document_generator.py`: Contains functions to generate PO and Packing Slip PDFs using ReportLab. Verified to correctly use descriptions passed from `app.py`.

* `email_service.py`: Contains `send_po_email` function using Postmark API.

* `.env`: CRITICAL & UNCOMMITTED. Stores all secrets, API keys, database connection details, Ship From address.

* `requirements.txt`: Lists Python dependencies.

Frontend (`src/` Directory - React/Vite):

* `main.jsx`: Entry point for the React application.

* `App.jsx`: Main application component, sets up routing using `react-router-dom`. Contains main navigation structure.

* `components/` or `pages/`:

  * `Dashboard.jsx`: Displays the list of orders.

    * **Recent Changes:** Status filtering, conditional display of customer comments, date/time moved to top-right of mobile cards, title styling, hydration error fix, "Status:" label on mobile badge overlap fix.

  * `OrderDetail.jsx`: Displays details of a single order and contains the interactive processing form.

    * **Recent Changes:** Displays HPE custom descriptions (with fallback logic), "RFQ Sent" status update on Brokerbin link click, refined form field disabling logic (fields active for "RFQ Sent", disabled for "Processed"), success/error messages moved above process button, "Back to Dashboard" link removed, "Order details not found" message flashing mitigated, AbortController logic for API calls.

  * `Suppliers.jsx`, `Products.jsx`: Placeholders for future development.

* `App.css`: Global styles. (Consider moving CSS variable definitions here from `OrderDetail.css`).

* `index.css`: Base styles, font imports.

* `Dashboard.css`, `OrderDetail.css`: Component-specific styles, updated for recent UI changes.

6. Database Schema

(Based on original Handover Report, Section 6 - Key tables listed, refer to original for full field lists)

* `orders`: Includes `customer_notes` (TEXT, NULLABLE) for BigCommerce customer messages.

* `order_line_items`

* `suppliers`: Includes `defaultPOnotes` (TEXT, NULLABLE).

* `hpe_part_mappings`: Maps BigCommerce SKU to HPE OptionPN. (Exists and Populated).

* `hpe_description_mappings`: Maps OptionPN to custom PO description. (Feature complete, table assumed created and populated).

* `purchase_orders`

* `po_line_items`: `sku` column stores the final HPE OptionPN (or fallback original SKU). `description` column stores the description from the frontend form (defaulted or manually edited).

* `shipments`

* `products`: Basic product info, used as a fallback for descriptions by `/api/lookup/description/<sku>`.

7. API Integrations & Credentials (.env variables)

****(Refer to original Handover Report, Section 7 for the full list of .env variables - Assumed Unchanged)****

* BigCommerce: `BIGCOMMERCE_STORE_HASH`, `BIGCOMMERCE_ACCESS_TOKEN`, `BC_PROCESSING_STATUS_ID`, `BC_SHIPPED_STATUS_ID`.

* UPS: `UPS_CLIENT_ID`, `UPS_CLIENT_SECRET`, `UPS_BILLING_ACCOUNT_NUMBER`. (Also `UPS_USERNAME`, `UPS_PASSWORD_UPS` if still used by `shipping_service.py` for any part of OAuth/API calls).

* Postmark: `EMAIL_API_KEY`, `EMAIL_SENDER_ADDRESS`, `EMAIL_BCC_ADDRESS`.

* Google Cloud Storage: `GCS_BUCKET_NAME`.

* Database (Cloud SQL): `DB_CONNECTION_NAME`, `DB_USER`, `DB_PASSWORD`, `DB_NAME`, `DB_DRIVER`.

* Ship From Address: `SHIP_FROM_...` variables.

* Flask Environment: `FLASK_ENV`, `FLASK_RUN_HOST`, `FLASK_RUN_PORT`.

8. Key Features Implemented / Progress Details (Consolidated)

(Combines original report Section 8 with recent developments)

* **Core Order Processing Workflow (Backend - `POST /api/orders/<id>/process`):**

  * Successfully orchestrates DB operations, PDF generation (PO, Packing Slip), UPS Label generation, GCS uploads, Postmark email, and BigCommerce shipment updates within a database transaction.

  * Correctly uses frontend-submitted descriptions for PO line items (DB and PDF).

  * Correctly determines and uses the mapped HPE Option PN (with underscore fallback logic) or the original SKU for PO line items.

* **HPE Custom Description & Fallback SKU Mapping Feature (Backend & Frontend):**

  * **COMPLETED.**

  * Backend (`get_order_details`, `process_order`, `get_hpe_mapping_with_fallback`) correctly identifies the HPE Option PN, including handling SKUs with underscores (e.g., `PART_OPTIONPN` by looking up `OPTIONPN`).

  * Backend fetches custom PO descriptions from `hpe_description_mappings` based on the determined Option PN.

  * Frontend (`OrderDetail.jsx`) defaults the "Description" field using this custom HPE description or the original item name.

  * The processing endpoint uses the (potentially user-edited) description from the frontend for the PO PDF and database.

* **Order Ingestion (`/ingest_orders`):**

  * Functional, fetches orders by status from BigCommerce.

  * Saves BigCommerce `customer_message` to local `orders.customer_notes`.

  * Bugs related to updating existing orders and attribute errors fixed.

* **Order Display & Interaction (Frontend):**

  * `Dashboard.jsx`: Displays orders with status filtering, responsive card/table views. Mobile cards now show date/time in top-right and conditional customer comments. Title styled.

  * `OrderDetail.jsx`:

    * Displays comprehensive order details, including original SKUs and mapped HPE PNs (with Brokerbin links).

    * Displays customer comments.

    * Interactive form for PO generation with dynamic description lookup for purchase SKUs.

    * "RFQ Sent" status update on Brokerbin link click is functional.

    * Form fields remain editable if status is "RFQ Sent"; disabled only if "Processed". "PROCESS ORDER" button disabled for "RFQ Sent" or "Processed".

    * Success/error messages for processing appear above the button.

* **Database Schema & Initial Data:** Core tables defined. `hpe_part_mappings` populated. `hpe_description_mappings` feature is complete (table assumed created/populated). `suppliers` table includes `defaultPOnotes`.

* **API Integrations:** UPS, BigCommerce, Postmark, GCS are all functionally integrated for the core processing flow.

9. Critical Next Steps / Remaining MVP Tasks (Updated Plan)

(Based on original Handover Report Section 9 and progress. "Task X" refers to original document numbering where applicable.)

1. **~~Implement HPE Custom Description Feature:~~**

  * **STATUS: COMPLETED.** (Covered creation of `hpe_description_mappings`, population, and full-stack testing).

2. **~~Test Frontend Manual Overrides:~~**

* **STATUS: COMPLETED.** (Verified that manually edited descriptions in `OrderDetail.jsx` are used for PO PDF and DB).

3. **Backend PO Export (`GET /api/exports/pos`):** (Original Task 13)

  * **STATUS: PENDING.**

  * **Goal:** Create an endpoint that generates and returns an Excel (.xlsx) file summarizing Purchase Order details.

  * **Action Items:**

    * Define the Flask route in `app.py`.

    * Write the SQL query to join `purchase_orders`, `po_line_items`, `suppliers`, and `orders` to gather all necessary data.

    * Use `openpyxl` (or similar Python library) to generate the .xlsx file in memory.

    * Use Flask's `send_file` to return the generated Excel file.

4. **Frontend Dashboard - PO Export Trigger:** (Original Task 12.2)

  * **STATUS: PENDING.**

  * **Goal:** Add a button/link on the `Dashboard.jsx` to trigger the PO export.

  * **Action Items:**

    * Add a UI element (e.g., "Export POs" button) to `Dashboard.jsx`.

    * Make this button trigger a `GET` request to `/api/exports/pos`.

    * Handle the file download response initiated by the browser.

    * Consider adding simple loading feedback on the frontend button.

5. **Deployment (GCP):** (Original Task 14)

  * **STATUS: PENDING.** This is a significant phase.

  * **Action Items:**

    * **Containerization:** Create `Dockerfile` for the Flask backend. (Original Task 14.1)

    * **Frontend Deployment Strategy:** Decide on frontend deployment (e.g., build static files served by Flask/Nginx, or separate frontend container/service on Cloud Run).

    * **Secret Manager Setup:** Configure GCP Secret Manager to securely store all `.env` variables. (Original Task 14.2)

    * **Cloud Run Service(s) Configuration:**

        * Create Cloud Run service(s) for backend (and frontend if applicable).

        * Configure environment variables to pull from Secret Manager.

        * Set up Cloud SQL Proxy sidecar or direct connection with IAM auth for database access.

        * Configure necessary IAM roles for the Cloud Run service account (Cloud SQL Client, Secret Manager Secret Accessor, GCS Object Admin, etc.). (Original Task 14.3)

    * **Networking:** Configure VPC connectors or firewall rules if needed. (Original Task 14.4)

    * **Deploy & Thorough Test:** Deploy to Cloud Run and perform comprehensive testing in the cloud environment. (Original Task 14.5)

6. **Comprehensive Testing:** (Original Task 15)

  * **STATUS: PARTIALLY DONE (developer testing), FORMAL TESTING PENDING.**

  * **Action Items:**

    * **End-to-End Testing:** Test full flows (Dashboard -> Order Detail -> Process -> Verify DB/GCS/Email/BC updates) for various scenarios (e.g., orders with direct HPE mapping, orders with underscore fallback SKU mapping, orders with no HPE mapping, orders with manual SKU/description overrides). (Original Task 15.1)

    * **Edge Case Testing:** Test API errors, missing mappings, validation failures, international order handling (should be blocked from auto-processing), network issues. (Original Task 15.2)

    * **Unit/Integration Tests (Recommended):** Add automated tests for critical backend logic (HPE mapping, SKU fallback, PO number generation, status updates, API client interactions). (Original Task 15.3)

7. **Postmark Account Approval:** (Original Task 6 in original outline)

  * **STATUS: PENDING VERIFICATION.**

  * **Action Item:** Ensure the Postmark account is fully approved for sending emails to external supplier domains before production deployment. This may require DNS changes.

8. **Supplier and Product (HPE Mapping) Management UI:** (Future Enhancement)

   * **STATUS: PENDING.** (Placeholder components `Suppliers.jsx`, `Products.jsx` exist).

   * **Goal:** Develop frontend interfaces for CRUD operations on suppliers and HPE part mappings (`hpe_part_mappings` table, potentially `hpe_description_mappings` as well).

   * **Action Items:** Design and implement these React components and connect them to the existing backend API endpoints.

9. **Font Customization:**

   * **STATUS: PENDING.** (User expressed desire for "Eloquia" font).

   * **Action Item:** Implement "Eloquia" font globally using `@font-face` or Google Fonts if available. Update `App.css` or `index.css`.

10. Detailed Task Breakdown (Elaboration on Next Steps)

* **(Tasks 3 & 4) PO Export Feature:**

   * **Backend:**

      * Design the Excel sheet layout: Determine columns needed (e.g., PO Number, PO Date, Supplier Name, Supplier Email, Original Order ID, Item SKU (HPE PN), Item Description, Item Qty, Item Unit Cost, Item Total, PO Total Amount, Payment Terms, Payment Instructions).

      * The SQL query will need joins across `purchase_orders`, `po_line_items`, `suppliers`, and `orders`.

      * Use `openpyxl` for creating the `.xlsx` file. Handle data types and formatting (dates, currency) appropriately for Excel.

      * Ensure Flask's `send_file` uses correct MIME type (`application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`) and `as_attachment=True` with a filename.

   * **Frontend:**

      * Add an "Export POs" button to `Dashboard.jsx`.

* On click, initiate a `GET` request to `/api/exports/pos`.

* The browser will handle the download. Provide user feedback (e.g., "Exporting..." message).

* Consider adding date range filters for the export if needed.

* **(Task 5) Deployment to GCP:**

* **Dockerfile (Backend):** Use a Python base image (e.g., `python:3.9-slim`). Copy application code, `requirements.txt`. Install dependencies. Expose the port Flask runs on (e.g., 8080). Use `gunicorn` as the WSGI server in the `CMD` or `ENTRYPOINT`.

* **Frontend Build:** If serving static files, `npm run build` in the frontend directory will create a `dist` folder. This can be served by the Flask app (using `send_from_directory` for `/` and static assets) or by a separate web server like Nginx in another container, or directly via Cloud Run if configured.

* **Secret Manager:** Create secrets in GCP Secret Manager for *all* variables currently in `.env`. Grant the Cloud Run service account the "Secret Manager Secret Accessor" IAM role.

* **Cloud Run Configuration:**

* Set appropriate CPU/memory, concurrency, min/max instances.

* Map environment variables in Cloud Run to the secrets in Secret Manager (e.g., `DB_USER` in Cloud Run gets value from secret `projects/YOUR_PROJECT/secrets/DB_USER/versions/latest`).

* **Cloud SQL Connection:** Use the Cloud SQL Auth Proxy as a sidecar container in your Cloud Run service. The application will connect to the proxy on `127.0.0.1:PROXY_PORT`. Alternatively, configure serverless VPC access for direct connection if preferred (more complex setup).

* **Service Account Permissions:** The Cloud Run service's runtime service account needs:

* `Cloud SQL Client` (to connect to Cloud SQL).

* `Secret Manager Secret Accessor` (to read secrets).

* `Storage Object Admin` or `Storage Object Creator/Viewer` on the GCS bucket (for document uploads/reads).

* Any other permissions needed for GCP services.

* **IAM & Firewall:** Ensure firewall rules for Cloud SQL allow connections from Cloud Run (typically handled by the proxy or private IP).

* **(Task 6) Comprehensive Testing:**

  * Define specific E2E test cases:

    * Ingest a new domestic order.

    * Open order, verify HPE mapping (direct and underscore fallback), verify custom description default.

    * Click Brokerbin link -> verify status changes to "RFQ Sent", verify form fields remain editable but Process button is disabled.

    * Manually edit a Purchase SKU and Description on an "RFQ Sent" order.

    * Process the "RFQ Sent" order: select supplier, fill costs, notes, weight.

    * Verify: PO PDF (correct SKU, correct edited/custom description, costs), Packing Slip, Label.

    * Verify: GCS uploads.

    * Verify: Email to supplier with attachments.

    * Verify: BigCommerce order updated (tracking, status).

    * Verify: Local DB status updated to "Processed".

    * Verify: `po_line_items` has correct SKU and description.

  * Test error conditions: API failures (UPS, BC, Postmark), GCS upload failure, database errors during processing (ensure rollback).

* **(Task 9) Font Customization:**

  * Obtain "Eloquia" font files (e.g., `.woff2`, `.woff`).

  * Place them in `frontend/src/assets/fonts/`.

  * Add `@font-face` rules to `frontend/src/App.css` or `frontend/src/index.css`.

  * Update the global `font-family` style on `body` to prioritize "Eloquia", with fallbacks like `Inter, system-ui, sans-serif`.

11. Known Issues / Considerations (Updated)

(Based on original Handover Report, Section 11, with updates)

* **UPS CIE Duplicate Tracking Numbers:** The UPS test (CIE) environment may return duplicate tracking numbers. The current `shipping_service.py` should ideally handle this by appending a unique suffix (e.g., timestamp, UUID snippet) to tracking numbers *only* when an environment variable indicates it's a test environment connected to UPS CIE. This prevents unique constraint errors in the `shipments` table during testing. (Verify implementation in `shipping_service.py`).

* **CSS Variable Scope:** CSS Variables are currently defined in `OrderDetail.css`. For better maintainability, consider moving these to a more global CSS file like `App.css` or `index.css` if they are used across multiple components.

* **`document_generator.py` Arguments:** While verified to work with current data, always double-check data structures passed if `po_items_for_pdf_generation` or `packing_slip_items` preparation logic in `app.py` changes significantly.

* **Error Handling Granularity:** Current error handling is functional. Future enhancements could include more granular error catching on the backend and more user-friendly, specific error messages on the frontend (e.g., distinguishing between a failed UPS call vs. a failed BigCommerce update). Implement more robust structured logging on the backend (e.g., using Python's `logging` module).

* **International Orders:** Currently blocked from automated processing by `process_order`. Future development will be needed to handle customs documentation, different shipping logic, etc.

* **Scalability:** The current synchronous design is suitable for moderate load. For very high volume, consider transitioning long-running tasks (PDF generation, external API calls within `process_order`) to an asynchronous task queue (e.g., Celery with Redis/RabbitMQ).

* **Frontend State Management for "Order Details Not Found":** While mitigated, a brief flash of "Order details not found" might still occur in `OrderDetail.jsx` due to the nature of asynchronous data loading and React's render cycle. Further refinement of loading states or initial component render logic might be needed if this remains an issue.

* **Security:** Ensure all API keys and sensitive credentials are *never* hardcoded and are managed securely (e.g., via `.env` locally and GCP Secret Manager in production). Review input validation on all API endpoints.

12. Environment Setup Guide

(Based on original Handover Report, Section 12 - Verify Python/Node versions if specific ones were used)

1. **Clone Repository / Obtain Files:**

   * Get the latest backend and frontend code (provided through uploads in this chat and contained within the immersives).

2. **Backend Setup (Python/Flask):**

   * Ensure Python 3.x is installed (e.g., Python 3.9+).

   * Navigate to the backend project root directory.

   * Create a Python virtual environment: `python -m venv venv`

   * Activate the environment:

      * Windows: `venv\Scripts\activate`

      * macOS/Linux: `source venv/bin/activate`

   * Install dependencies: `pip install -r requirements.txt`

   * Create a `.env` file in the backend root directory. Populate it with all required variables listed in Section 7 (Database, APIs, Ship From, GCS Bucket, etc.). **DO NOT COMMIT THIS FILE.**

   * **Cloud SQL Connection (Local Development):**

      * Install the `gcloud` CLI and authenticate (`gcloud init`, `gcloud auth application-default login`).

      * Download and run the Cloud SQL Auth Proxy executable for your OS, pointing to your Cloud SQL instance connection name (e.g., `cloud-sql-proxy <INSTANCE_CONNECTION_NAME>`).

      * Ensure your `.env` file has `DB_DRIVER=pg8000` (or your preferred driver if you changed it) and other DB variables set correctly for proxy connection (host usually `127.0.0.1`, port `5432`).

  * Run the Flask app: `flask run` (or `python app.py`). It typically runs on `http://127.0.0.1:8080` (check `.env` for `FLASK_RUN_HOST`/`FLASK_RUN_PORT` or terminal output).

3. **Frontend Setup (React/Vite):**

  * Ensure Node.js (e.g., LTS version) and npm (or yarn) are installed.

  * Navigate to the frontend project root directory.

  * Install dependencies: `npm install` (or `yarn install`).

  * Run the Vite dev server: `npm run dev` (or `yarn dev`).

  * Access the app in your browser, typically at `http://localhost:5173` (check terminal output).

  * **API Proxy/CORS:** The `vite.config.js` should have a proxy set up for `/api` requests to be forwarded to the backend server (e.g., `http://localhost:8080`) to avoid CORS issues during development. (Verify this configuration).

13. Code Access

The latest versions of the key frontend components (`OrderDetail.jsx`, `Dashboard.jsx`, `Dashboard.css`) and the backend application (`app.py`) are available within the immersive documents generated during our collaborative session. The initial project files (including other backend services and frontend CSS) were provided by you. Ensure all these pieces are consolidated in your version control system.

14. Conclusion

Significant progress has been made on the G1 PO App. The core backend processing workflow is robust, including enhanced HPE part mapping with SKU fallback and correct description handling. Key API integrations are functional. The frontend provides essential viewing, filtering, and interaction capabilities for order processing, with several UI/UX improvements implemented.

The application is now in a more advanced state, with the "HPE Custom Description" feature fully completed and several other enhancements and fixes in place. The next critical phases involve implementing the PO export feature, deploying the application to GCP, and conducting thorough end-to-end testing. This updated document provides the necessary context, current status, and a refined roadmap for the incoming developer to successfully complete the MVP and move towards production.