Project Handover Report: G1 PO App

Date: May 8, 2025 (Adjust as needed)

Prepared For: Incoming Developer

Prepared By: Mark (via collaboration with AI Assistant)

1. Project Overview

- **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.
    - Allow users to review orders and trigger processing.
    - Generate Purchase Orders (PDF) using specific HPE Option PNs mapped from original BigCommerce SKUs.
    - Generate Packing Slips (PDF).
    - Generate UPS Shipping Labels (PDF) via UPS API, handling OAuth.
    - Email generated PO, Packing Slip, and Label to the selected supplier via Postmark API.
    - Upload generated documents to Google Cloud Storage (GCS).
    - Update the order status and add tracking information in BigCommerce via API.
    - Update the local database status for processed orders.
    - Provide a web interface (React/Vite) for viewing orders, managing suppliers/products (future), and initiating the processing workflow.

2. Current Status (As of Handover)

- **Backend:**
    - The core POST /api/orders/<id>/process endpoint is largely functional, integrating database operations, document generation, UPS label creation, GCS upload, Postmark email, and BigCommerce updates within a single transaction.

    - HPE Part Number mapping logic (joining order_line_items with hpe_part_mappings) is implemented in both GET /api/orders/<id> and POST /api/orders/<id>/process. It correctly uses the hpe_option_pn when found and falls back to the original_sku.
    - A separate hpe_description_mappings table and logic for custom PO descriptions based on hpe_option_pn have been designed (requires data population and final testing).
    - Backend endpoint GET /api/orders supports filtering by status (?status=...).
    - Backend endpoint POST /api/orders/<id>/status created for specific status updates (e.g., "RFQ Sent").
    - Backend endpoint GET /api/lookup/description/<sku> created for dynamic frontend description lookups.
    - Database connection (Cloud SQL via proxy locally) is established.
    - Basic error handling and transaction rollback are in place.
- **Frontend:**
    - Basic project structure (main.jsx, App.jsx) and routing (react-router-dom) exist.
    - Placeholder components (Suppliers.jsx, Products.jsx) exist.
    - Dashboard.jsx: Fetches and displays orders, implements status filtering (defaulting to 'new'), includes responsive "card" layout for mobile, makes cards clickable to navigate to order details.
    - OrderDetail.jsx: Fetches and displays detailed order information. Implemented as an interactive form for processing:
        - Displays mapped hpe_option_pn / custom description alongside original SKU/name.
        - Allows supplier selection (autofills PO Notes from suppliers.defaultPOnotes).
        - Allows editing of Purchase SKU, PO Description (as <textarea>), Quantity, and Unit Cost for each line item before processing.
        - Includes dynamic description lookup based on Purchase SKU input (debounced).

        - Includes Brokerbin links for SKUs; clicking these links triggers a status update to "RFQ Sent" via the backend API.
        - Handles form submission, validation, loading/processing states, and success/error feedback.
        - Form is disabled if the order status is 'Processed' or 'RFQ Sent'.
        - Includes responsive layout adjustments.
    - CSS styling applied to App.css, Dashboard.css, OrderDetail.css for layout, responsiveness, and basic visual appeal (including light blue mobile cards on Dashboard).
- **Database:**
    - Core tables (orders, order_line_items, suppliers, purchase_orders, po_line_items, shipments, products) schema defined and assumed to exist in Cloud SQL PostgreSQL.
    - hpe_part_mappings table created and populated (approx. 8000 rows migrated).
    - hpe_description_mappings table designed (requires creation and population).
    - suppliers table updated with defaultPOnotes column.
- **Integrations:**
    - UPS API: OAuth and Label Generation (Shipment request) working in CIE test environment.
    - BigCommerce API: Order fetching (basic), Shipment creation/status update working.
    - Postmark API: Email sending with attachments working (tested to internal domain).
    - Google Cloud Storage: Document upload working.

3. Technology Stack

- **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 dependencies.

4. Architecture Overview

- **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)

- **Backend (Root Directory):**
  - app.py: Main Flask application file. Initializes DB engine, GCS client. Defines all API routes (/, /api/orders, /api/orders/<id>, /api/orders/<id>/process, /api/orders/<id>/status, /api/suppliers, /api/products, /api/lookup/description/<sku>, /ingest_or

    ders, /api/exports/pos [planned]). Contains primary orchestration logic for processing.

    - shipping_service.py: Contains functions for UPS OAuth, UPS label generation (including state mapping), and BigCommerce order updates (shipment creation).
    - document_generator.py: Contains functions to generate PO and Packing Slip PDFs using ReportLab. *(Needs review for argument signatures vs current usage)*.
    - email_service.py: Contains send_po_email function using Postmark API and attachments.
    - .env: **CRITICAL & UNCOMMITTED.** Stores all secrets, API keys, database connection details, Ship From address.
    - requirements.txt: Lists Python dependencies.
    - helpers.py (or similar, possibly within app.py): Contains utility functions like convert_row_to_dict, make_json_safe.
- **Frontend (src/ Directory):**
    - main.jsx: Entry point for the React application.
    - App.jsx: Main application component, likely sets up routing using react-router-dom. Contains main navigation structure.
    - components/ (or similar): Directory for reusable UI components.
    - pages/ (or similar, maybe directly in src/):
        - Dashboard.jsx: Displays the list of orders, includes filtering and responsive layout.
        - OrderDetail.jsx: Displays details of a single order and contains the interactive processing form.
        - Suppliers.jsx (Placeholder)
        - Products.jsx (Placeholder)
    - App.css: Global styles, including #root modifications and potentially shared component styles.
    - index.css: Often contains base styles and font imports.

- Dashboard.css: Component-specific styles.
- OrderDetail.css: Component-specific styles.

6. Database Schema

(Assumes tables exist in Cloud SQL PostgreSQL instance)

- **orders**: Ingested order details.
    - id (PK, SERIAL), bigcommerce_order_id (INTEGER, UNIQUE), customer_name, customer_company (VARCHAR, NULLABLE), customer_shipping_... fields (address, phone, email), order_date (TIMESTAMPTZ), total_sale_price (DECIMAL), status (VARCHAR - e.g., 'new', 'Processed', 'RFQ Sent', 'international_manual'), is_international (BOOLEAN), customer_shipping_method (VARCHAR), payment_method (VARCHAR), created_at, updated_at (TIMESTAMPTZ).
- **order_line_items**: Original line items from BigCommerce.
    - id (PK, SERIAL), order_id (FK to orders.id), bigcommerce_line_item_id (INTEGER), sku (VARCHAR - **Original BigCommerce SKU**), name (TEXT), quantity (INTEGER), sale_price (DECIMAL), created_at, updated_at (TIMESTAMPTZ).
- **suppliers**: Supplier details.
    - id (PK, SERIAL), name (VARCHAR), email (VARCHAR, UNIQUE), address_... fields, payment_terms (VARCHAR), defaultPOnotes (TEXT, NULLABLE), created_at, updated_at (TIMESTAMPTZ).
- **hpe_part_mappings**: Maps BigCommerce SKU to HPE OptionPN. *(Exists and Populated)*
    - sku (VARCHAR(100) PRIMARY KEY or UNIQUE NOT NULL - **Original BigCommerce SKU**), option_pn (VARCHAR(100) NOT NULL), pn_type (VARCHAR(50) NULLABLE).
- **hpe_description_mappings**: Maps OptionPN to custom PO description. *(Designed, needs creation/population)*
    - option_pn (VARCHAR(100) PRIMARY KEY), po_description (TEXT NOT NULL), created_at, updated_at (TIMESTAMPTZ, optional).

- **purchase_orders**: Stores generated POs.
    - id (PK, SERIAL), po_number (INTEGER, UNIQUE NOT NULL), order_id (FK to orders.id), supplier_id (FK to suppliers.id), po_date (TIMESTAMPTZ), payment_instructions (TEXT), status (VARCHAR NOT NULL DEFAULT 'New'), total_amount (DECIMAL), po_pdf_gcs_path (VARCHAR(512) NULL), created_at, updated_at (TIMESTAMPTZ).
- **po_line_items**: Stores items on a specific PO.
    - id (PK, SERIAL), purchase_order_id (FK to purchase_orders.id), original_order_line_item_id (FK to order_line_items.id, NULLABLE), sku (VARCHAR - **Stores HPE OptionPN or fallback Original SKU**), description (TEXT - Stores mapped description or original name), quantity (INTEGER), unit_cost (DECIMAL), condition (VARCHAR), created_at, updated_at (TIMESTAMPTZ).
- **shipments**: Stores shipment details.
    - id (PK, SERIAL), purchase_order_id (FK to purchase_orders.id), tracking_number (VARCHAR, UNIQUE NOT NULL), shipping_method_name (VARCHAR(255) NULL), weight_lbs (DECIMAL or FLOAT), label_gcs_path (VARCHAR(512) NULL), packing_slip_gcs_path (VARCHAR(512) NULL), created_at (TIMESTAMPTZ).
- **products**: Basic product info (may be superseded by mappings).
    - id (PK, SERIAL), sku (VARCHAR, UNIQUE), standard_description (TEXT), created_at, updated_at (TIMESTAMPTZ).

7. API Integrations & Credentials (.env variables)

- **BigCommerce:** BIGCOMMERCE_STORE_HASH, BIGCOMMERCE_ACCESS_TOKEN, BC_PROCESSING_STATUS_ID (optional, for explicit status update), BC_SHIPPED_STATUS_ID. (Uses V2 REST API).
- **UPS:** UPS_CLIENT_ID, UPS_CLIENT_SECRET, UPS_BILLING_ACCOUNT_NUMBER. (Uses JSON REST Shipping API v2409 via CIE test env). **Note:** Test env returns duplicate tracking numbers.

- **Postmark:** EMAIL_API_KEY (Server Token), EMAIL_SENDER_ADDRESS (Verified Signature), EMAIL_BCC_ADDRESS (Optional). **Note:** Account requires approval for sending outside sender domain.
- **Google Cloud Storage:** GCS_BUCKET_NAME. Requires IAM permissions for credentials used by app.py (ADC locally, service account in Cloud Run).
- **Database (Cloud SQL):** DB_HOST (for proxy, e.g., 127.0.0.1), DB_PORT (e.g., 5432), DB_USER, DB_PASSWORD, DB_NAME. (Uses pg8000 driver).
- **Ship From Address:** SHIP_FROM_NAME, SHIP_FROM_CONTACT, SHIP_FROM_STREET1, SHIP_FROM_STREET2, SHIP_FROM_CITY, SHIP_FROM_STATE, SHIP_FROM_ZIP, SHIP_FROM_COUNTRY, SHIP_FROM_PHONE.
- **(Optional) Environment Flag:** FLASK_ENV=development or IS_TEST_ENVIRONMENT=true (useful for differentiating UPS tracking # logic).

8. Key Features Implemented / Progress Details

- **Database Schema & Initial Data:** Core tables defined, hpe_part_mappings table created and populated. suppliers table updated with defaultPOnotes.
- **Backend Order Processing (POST /api/orders/<id>/process):**
  - Successfully orchestrates steps 1-14 within a DB transaction.
  - Fetches order, supplier, and *line item details including joined HPE mapping data*.
  - Handles domestic check.
  - Generates sequential PO numbers.
  - Inserts purchase_orders record.
  - Correctly looks up hpe_option_pn and custom hpe_po_description (if available).
  - Correctly uses hpe_option_pn (or original_sku as fallback) for the sku column when inserting into po_line_items.
  - Correctly uses the custom or original description when inserting into po_line_items.

    - Prepares data for PDF generation using the correct final SKU and Description.
    - Generates PO & Packing Slip PDFs (via document_generator.py).
    - Generates UPS Label & Tracking Number (via shipping_service.py, handles OAuth).
    - Inserts shipments record (handles potential duplicate tracking # from UPS CIE).
    - Uploads all 3 PDFs to GCS under processed_orders/order_<id>/....
    - Updates purchase_orders and shipments tables with GCS paths.
    - Sends email with all 3 PDF attachments to supplier via Postmark (via email_service.py).
    - Updates BigCommerce order by creating a shipment with tracking number (implicitly updates status to Shipped).
    - Updates local orders table status to 'Processed'.
    - Commits transaction on success, rolls back on any failure.
- **Backend Order Fetching (GET /api/orders, GET /api/orders/<id>):**
    - /api/orders fetches list, supports ?status= filtering, sorts by date.
    - /api/orders/<id> fetches single order and its line items, correctly performing joins to hpe_part_mappings and hpe_description_mappings and returning data using explicit aliases
      (e.g., original_sku, hpe_option_pn, hpe_po_description).
- **Backend Supplier Fetching (GET /api/suppliers):**
    - Fetches list of suppliers including defaultPOnotes.
- **Backend Status Update (POST /api/orders/<id>/status):**
    - Allows updating only the status (e.g., to 'RFQ Sent').
- **Backend Description Lookup (GET /api/lookup/description/<sku>):**
    - Provides description based on SKU/OptionPN for frontend dynamic updates.
- **Frontend Dashboard (Dashboard.jsx):**
    - Displays orders fetched from /api/orders.

  - Implements status filter dropdown, defaulting to 'new'. Refetches data when filter changes.
  - Responsive layout: Desktop table view, Mobile "card" view (light blue background).
  - Mobile view displays Order#, Status, Order Date, Customer, Ship Method, Ship To, Total per card.
  - Entire mobile card is clickable, navigating to the Order Detail page.
- **Frontend Order Detail (OrderDetail.jsx):**
  - Fetches and displays comprehensive order details from /api/orders/<id>.
  - Displays original line items, showing original SKU (Brokerbin link) and distinct mapped OptionPN (Brokerbin link).
  - Provides interactive form for PO generation:
    - Supplier dropdown (populates PO Notes from defaultPOnotes).
    - Editable PO Notes textarea.
    - Editable list of "Items to Purchase":
      - Defaults Purchase SKU to hpe_option_pn or original_sku.
      - Defaults Description to hpe_po_description or line_item_name.
      - Allows editing Purchase SKU, Description (as textarea), Qty, Unit Cost.
      - Description dynamically updates (debounced) when Purchase SKU is changed (via /api/lookup/description).
    - Shipment Method dropdown (defaults based on customer method, maps Free Shipping).
    - Shipment Weight input.
  - Clicking Brokerbin links triggers status update to 'RFQ Sent' if order status is 'new'.
  - Form submission (POST /api/orders/<id>/process) with validation, loading state, success/error feedback.

    - Form disabled if order status is 'Processed' or 'RFQ Sent'.
    - Responsive layout for form elements.
- **Styling (App.css, Dashboard.css, OrderDetail.css):**
    - CSS Variables defined.
    - Global styles applied to #root for desktop centering and mobile full-width.
    - Component-specific styles applied for layout, cards, forms, buttons, badges, responsive behavior.

9. Critical Next Steps / Remaining MVP Tasks

(Based on original document Section 11 and progress)

1. **Implement HPE Custom Description Feature:**
    - Create the hpe_description_mappings table in PostgreSQL.
    - Populate the table with desired OptionPN-to-Description mappings.
    - Thoroughly test that GET /api/orders/<id> returns the hpe_po_description.
    - Test that OrderDetail.jsx defaults the description field correctly.
    - Test that POST /api/orders/<id>/process uses the correct description in po_line_items and the generated PO PDF.
2. **Test Frontend Manual Overrides:**
    - Thoroughly test editing the "Purchase SKU" and "Description" fields on the OrderDetail form.
    - Verify the POST /process endpoint receives the *overridden* values in the payload.
    - Verify the po_line_items table and the generated PO PDF reflect these manual overrides.
3. **Backend PO Export (GET /api/exports/pos):** (Task 13)
    - Define the Flask route.
    - Write the SQL query to join purchase_orders, po_line_items, suppliers, orders to get all necessary data for export.

- Use openpyxl library to generate an .xlsx file in memory.
- Use Flask's send_file to return the generated Excel file to the user.

4. **Frontend Dashboard - PO Export Trigger:** (Task 12.2)
    - Add a button/link to the Dashboard UI.
    - Make this button trigger a GET request to /api/exports/pos.
    - Handle the file download response initiated by the browser.
5. **Deployment (GCP):** (Task 14)
    - **Containerization (Task 14.1):** Create Dockerfile for the Flask backend. Decide on frontend deployment (build static files to serve via Flask/Nginx, or separate frontend container/service).
    - **Secret Manager Setup (Task 14.2):** Configure Secret Manager in GCP to securely store all .env variables.
    - **Cloud Run Service(s) Configuration (Task 14.3):** Create Cloud Run service(s). Configure environment variables to pull from Secret Manager. Set up DB connection (using Cloud SQL Proxy sidecar or direct connection with IAM auth). Configure necessary IAM roles for the service account (Cloud SQL Client, Secret Manager Secret Accessor, GCS Object Admin, etc.).
    - **Networking (Task 14.4):** Configure VPC connectors or firewall rules if needed (e.g., for Cloud SQL connection or if services are internal).
    - **Deploy & Test (Task 14.5):** Deploy the container(s) to Cloud Run and perform thorough testing in the cloud environment.
6. **Testing:** (Task 15)
    - **End-to-End Testing (Task 15.1):** Test full flows (Dashboard -> Order Detail -> Process -> Verify DB/GCS/Email/BC).
    - **Edge Case Testing (Task 15.2):** Test API errors, missing mappings, validation failures, international order handling (should be blocked), network issues.
    - **Unit/Integration Tests (Task 15.3 - Recommended):** Add tests for critical backend logic (HPE mapping lookup, PO number generation, status updates, API client interactions).
7. **Postmark Account Approval:** (Task 6 in original doc)

    - Ensure the Postmark account is fully approved for sending emails to external supplier domains before production deployment.

10. Detailed Task Breakdown (Elaboration on Next Steps)

- **(Task 1/2) HPE Desc Mapping & Overrides:** Focus on data integrity and ensuring the correct description/SKU is used based on the priority: Frontend Override > Mapped Custom Description/OptionPN > Original Name/SKU. Requires careful testing of process_order Step 7 logic and PDF output.
- **(Task 3/4) PO Export:** Design the Excel sheet layout. The SQL query needs careful joins. Consider date/number formatting in Excel using openpyxl. Ensure appropriate headers are sent in the Flask response for send_file. Add simple loading feedback on the frontend button.
- **(Task 5) Deployment:** This is a significant phase.
    - *Dockerfile:* Choose a Python base image, copy code, install requirements.txt, expose port, define CMD or ENTRYPOINT (e.g., using gunicorn).
    - *Secret Manager:* Create secrets for each .env variable. Grant access to the Cloud Run service account.
    - *Cloud Run Config:* Pay attention to CPU/memory allocation, concurrency, environment variables (mapping secrets), Cloud SQL connection method (Proxy sidecar is often recommended for security/ease), service account permissions.
    - *Frontend Build (if serving static):* Add build step (npm run build) and configure Flask/Nginx to serve the dist folder.
- **(Task 6) Testing:** Define specific E2E test cases (e.g., "Process domestic order with full HPE mapping", "Process domestic order with fallback SKU", "Process domestic order with manual SKU/Desc override", "Click Brokerbin link, verify status change", "Filter dashboard by 'Processed'").
- **(Task 7) Postmark:** Check Postmark dashboard for sending domain approval status. May require DNS changes.

11. Known Issues / Considerations

- **UPS CIE Duplicate Tracking Numbers:** The UPS test environment returns duplicate tracking numbers, causing unique constraint errors in
  the shipments table. **Recommendation:** Implement logic (likely

in shipping_service.py or app.py Step 9/10) to append a unique suffix (e.g., timestamp, UUID snippet) to tracking numbers *only* when running in a non-production environment connected to UPS CIE.

- **CSS Consolidation:** Currently, common styles (like .card) might be defined in multiple places (App.css, Dashboard.css, OrderDetail.css). Refactoring to define common styles globally (App.css or index.css) and keeping component CSS specific would be cleaner.
- **document_generator.py Arguments:** The initial outline mentioned potential ambiguity in arguments passed. While the current implementation seems to pass the correct data (po_items_for_pdf_generation), a quick review of the generate_purchase_order_pdf function signature and usage is advised.
- **Error Handling:** Current error handling is basic. Consider adding more granular error catching and more user-friendly error messages on the frontend. Implement more robust logging on the backend.
- **International Orders:** Currently blocked by process_order. Future enhancement needed for handling customs, different shipping logic, etc.
- **Scalability:** Current design is suitable for moderate load. For very high volume, consider asynchronous task queues (like Celery) for long-running processes (PDF generation, API calls).

12. Environment Setup Guide

1. **Clone Repository / Unzip Files:** Obtain the project code.
2. **Backend Setup:**
   - Ensure Python 3.x is installed.
   - 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 root directory. Populate it with *all* required variables listed in Section 7 (Database, APIs, Ship From, etc.). **DO NOT COMMIT THIS FILE.**
   - **Cloud SQL Connection:**
     - Install the gcloud CLI and authenticate (gcloud init, gcloud auth application-default login).
     - Download and install the Cloud SQL Auth Proxy executable for your OS.
     - Run the proxy in a separate terminal, pointing to the Cloud SQL instance connection name (provided in .env documentation or previous logs): cloud-sql-proxy <INSTANCE_CONNECTION_NAME>
     - Ensure your .env file has DB_HOST=127.0.0.1 and DB_PORT=5432 (or the proxy port).
   - Run the Flask app: flask run (or python app.py). By default, it should run on port 5000 or 8080 (check app.py or Flask output).
3. **Frontend Setup:**
   - Ensure Node.js and npm are installed.
   - Navigate to the frontend project root directory.
   - Install dependencies: npm install
   - Run the Vite dev server: npm run dev
   - Access the app in your browser, typically at http://localhost:5173 (check terminal output for the exact URL).
   - **API Proxy/CORS:** Ensure vite.config.js has a proxy configuration set up to forward requests from /api/* (or similar) to the backend server's address (e.g., http://localhost:8080) to avoid CORS errors during development.

13. Assets / Code Handover

- The code is provided via [Specify Method: e.g., shared Google Drive links to zip files, Git repository URL (mention branch if not main/master)].
- Backend code: [Link/Path to backend zip/repo]
- Frontend code: [Link/Path to frontend zip/repo]

- This handover document.
- The initial project outline document.

14. Conclusion

Significant progress has been made on the G1 PO App. The core backend processing workflow is functional, key API integrations are working, and the frontend provides essential viewing and interaction capabilities with responsive design improvements. The next phases involve completing the custom description mapping, implementing the PO export feature, thorough testing, and final deployment to GCP. This document provides the necessary context and a roadmap for the incoming developer to successfully complete the MVP.