G1 PO App - Comprehensive Handover Report & Next Steps

Date: May 11, 2025 (Reflecting current progress)
Prepared For: Incoming Developer
Prepared By: Mark (via collaboration with AI Assistant Gemini)

1. 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, and further consolidated May 9, 2025) with all subsequent development progress, features implemented, bug fixes addressed, and infrastructure setup up to May 11, 2025. The goal is to provide a clear and detailed guide for the incoming developer to understand the project's current state, architecture, codebase, deployment, and the remaining tasks to achieve the Minimum Viable Product (MVP) and future enhancements.

2. Project Overview (Recap from original report)

- **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 (Original Intent):**
    - 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 (including logic for underscore SKUs).
    - Generate Packing Slips (PDFs).
    - Generate UPS Shipping Labels (PDFs) via UPS API (including OAuth 2.0).
    - Email 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 order status and add tracking information in BigCommerce via its API.

- Update local application database status for processed orders.
- Provide a web interface (React/Vite) for viewing/filtering orders, viewing details, interactively preparing POs, and triggering processing.
- (Future) Manage suppliers and product mappings.

3. Current Status (As of May 11, 2025 - Incorporating Chat Progress)

The application has reached a significant stage of functionality, with the backend deployed to Google Cloud Run and the frontend deployed to Firebase Hosting.

- **Backend (Python/Flask - app.py and service modules):**
  - **Deployment:** Successfully Dockerized and deployed to **Google Cloud Run**.
    - Service Name: g1-po-app-backend-service
    - Region: us-central1 (as per current configuration)
    - URL: https://g1-po-app-backend-service-992027428168.us-central1.run.app
  - **Database:** Connected to a **PostgreSQL instance on Google Cloud SQL**.
    - Instance Connection Name: order-processing-app-458900:us-central1:order-app-db
    - Connection is managed via the Cloud SQL Auth Proxy sidecar in Cloud Run.
  - **Secrets Management:** All sensitive credentials (DB, BigCommerce, UPS, Postmark, GCS) are managed via **Google Cloud Secret Manager** and mapped as environment variables to the Cloud Run service.
  - **Order Ingestion (/ingest_orders - POST):**
    - Functional: Fetches orders from BigCommerce based on a configurable status ID (currently '7').
    - Saves/updates orders and line items in the local PostgreSQL database.
    - Correctly determines international status.
    - Saves BigCommerce customer_message to orders.customer_notes.
    - Error handling improved.

- **Order Retrieval (/api/orders, /api/orders/<id> - GET):** Functional, including filtering by status and fetching line item details with HPE part mapping.
- **HPE Part Number & Custom Description Mapping:** Robust logic (direct and underscore fallback) for hpe_option_pn and fetching hpe_po_description is implemented.
- **Order Processing (/api/orders/<id>/process - POST):**
    - Core workflow is largely functional within a single database transaction.
    - Generates sequential PO numbers.
    - **Document Generation:**
        - PO PDF: Generated by document_generator.py.
        - Packing Slip PDF: Generated by document_generator.py.
        - **UPS Label:**
            - Successfully integrates with **UPS Production API** (OAuth for token, Shipping API for label).
            - Requests label as a **GIF** image.
            - **Converts the GIF label to PDF** using Pillow and ReportLab. The PDF label is placed at the top of a US Letter page with 1-inch margins.
    - **Google Cloud Storage (GCS):** Uploads PO (PDF), Packing Slip (PDF), and the converted Label (PDF) to GCS under processed_orders/order_<id>/....
    - **Email (Postmark):**
        - Sends an email to the supplier with the PO (PDF), Packing Slip (PDF), and converted Label (PDF) as attachments.
        - Sends a separate email with structured PO data for QuickBooks import.
    - **BigCommerce Update:** Updates the BigCommerce order by creating a shipment with the tracking number.

        - **Local Status Update:** Updates the local orders table status to 'Processed'.
    - **Status Update (/api/orders/<id>/status - POST):** Functional for frontend-driven status changes (e.g., 'RFQ Sent').
    - **SKU/Description Lookup (/api/lookup/description/<sku> - GET):** Enhanced with fallback logic.
    - **Supplier and Product Mapping Endpoints (/api/suppliers, /api/products):** Basic CRUD exists (as per original report).
    - **CORS:** Configured in app.py using Flask-CORS to allow requests from the Firebase Hosting frontend URL (https://g1-po-app-77790.web.app).
- **Frontend (React/Vite - Dashboard.jsx, OrderDetail.jsx, etc.):**
    - **Deployment:** Built for production and deployed to **Firebase Hosting**.
        - Default URL: https://g1-po-app-77790.web.app
    - **API Communication:** Configured via .env.production (VITE_API_BASE_URL) to point to the live Cloud Run backend URL. All API calls in components have been updated to use this base URL.
    - **Dashboard (Dashboard.jsx):** Fetches and displays orders, status filtering works. Mobile card view and desktop table view implemented.
    - **Order Detail (OrderDetail.jsx):**
        - Fetches and displays comprehensive order details.
        - Interactive processing form allows supplier selection, PO notes editing, purchase item SKU/description/qty/cost editing.
        - Dynamic description lookup for purchase SKUs.
        - "RFQ Sent" logic on Brokerbin link click is functional.
        - Form disabling logic for processed orders.
        - "PROCESS ORDER" button triggers the backend processing workflow.
        - Most UI/UX enhancements from the original report (e.g., title styling, mobile card view improvements) are implemented.

    - **Supplier & Product Mapping Pages/Forms:** Placeholders and basic forms exist; functionality for listing, adding, editing suppliers and product mappings (standard descriptions) is present. The API calls in these components have also been updated to use VITE_API_BASE_URL.
    - **Font:** "Eloquia" font files are included in the project. The application is configured to use Eloquia, but there was a persistent issue getting it to render on the deployed site (browser not attempting to load font files). This needs re-investigation.
- **Database (PostgreSQL on Cloud SQL):**
    - Core tables defined and populated as needed (hpe_part_mappings, hpe_description_mappings feature complete).
    - Includes customer_notes in orders and defaultPOnotes in suppliers.
- **Integrations:**
    - **UPS API:** Functionally integrated with **Production environment** for OAuth and Label Generation (GIF requested, then converted to PDF).
    - **BigCommerce API:** Order fetching and Shipment creation/status update working.
    - **Postmark API:** Email sending with PDF/GIF attachments working.
    - **Google Cloud Storage (GCS):** Document upload (PO, Packing Slip, Label PDFs) working.
- **Scheduling (Order Ingestion):**
    - **Google Cloud Scheduler** job configured to call the /ingest_orders endpoint via HTTP POST every 5 minutes.
    - Currently uses "No auth" for the scheduler target, relying on the Cloud Run service being --allow-unauthenticated.

4. Technology Stack (Recap & Confirmations)

- **Backend:** Python 3.9+, Flask, SQLAlchemy, pg8000 (PostgreSQL driver), requests, google-cloud-sql-connector, google-cloud-storage, reportlab, Pillow, Flask-CORS, postmarker.
- **Frontend:** React, Vite, react-router-dom, axios (or fetch).

- **Database:** PostgreSQL (hosted on Google Cloud SQL).
- **Cloud Platform (GCP):**
    - Cloud Run (for backend hosting)
    - Cloud SQL (for PostgreSQL database)
    - Artifact Registry (for Docker images)
    - Secret Manager (for credentials)
    - Google Cloud Storage (GCS) (for document storage)
    - Cloud Scheduler (for cron jobs)
    - Firebase Hosting (for frontend hosting)
- **APIs:** UPS (Shipping, OAuth), BigCommerce (V2 REST), Postmark (Email).
- **Document Generation:** ReportLab (for POs, Packing Slips), Pillow + ReportLab (for UPS Label GIF to PDF conversion).
- **Environment:**
    - Backend: Python venv, .env file for local config, requirements.txt.
    - Frontend: Node.js/npm, .env.production for Vite, package.json.

5. Codebase Structure (Key Files & Recent Changes - Summary)

- **Backend (Python - order-processing-app directory):**
    - app.py: Main Flask app, API routes, core orchestration. CORS initialized here.
    - shipping_service.py: UPS API logic (OAuth, label generation, GIF-to-PDF conversion). Configured for switchable Test/Production UPS endpoints via UPS_API_ENVIRONMENT env var.
    - document_generator.py: PDF generation for POs and Packing Slips.
    - email_service.py: Postmark email sending logic for supplier POs and QuickBooks data. Uses postmarker library.
    - Dockerfile: Defines the Docker image for the backend. Uses gunicorn as WSGI server.
    - entrypoint.sh: Shell script used by Docker ENTRYPOINT to correctly launch Gunicorn with environment variables.

    - .dockerignore: Configured to exclude venv, __pycache__, backups, .git, etc.
    - requirements.txt: Lists all Python dependencies (including Flask-CORS, postmarker, Pillow).
    - .env (local only, in .gitignore): Stores local development secrets.
- **Frontend (React/Vite - order-processing-app-frontend directory):**
    - src/main.jsx: Entry point, imports global CSS (App.css, index.css), sets up BrowserRouter.
    - src/App.jsx: Main app component, defines routes, navigation. Imports App.css.
    - src/App.css: Global styles, @font-face for Eloquia, CSS variables for light/dark mode.
    - src/components/: Contains Dashboard.jsx, OrderDetail.jsx, SupplierList.jsx, etc. All API calls within these components have been updated to use import.meta.env.VITE_API_BASE_URL.
    - public/: Should contain static assets like favicon.ico. (If Eloquia fonts were moved here, this needs to be noted).
    - .env.production: Defines VITE_API_BASE_URL pointing to the live Cloud Run backend.
    - firebase.json, .firebaserc: Firebase Hosting configuration files.
    - vite.config.js: Vite configuration, including proxy for local development.

6. API Integrations & Credentials (.env / Secret Manager Variables)

- All sensitive credentials are now managed in Google Cloud Secret Manager for the deployed Cloud Run service.
- Local development uses an .env file in the backend root.
- Key variables include those for BigCommerce, UPS (Client ID, Secret, Account Number, a production environment flag), Postmark, GCS, Database, and Ship From address.
- A comprehensive list of required variables was assembled during deployment.

7. Key Features Implemented / Progress Details (Summary)

- Core Order Processing Workflow (Backend): Deployed and functional.
- HPE Custom Description & Fallback SKU Mapping: Implemented.
- Order Ingestion: Functional.
- Order Display & Interaction (Frontend): Deployed and largely functional.
- Database Schema & Initial Data: Defined and working.
- API Integrations (UPS, BigCommerce, Postmark, GCS): All functionally integrated for the core processing flow, with UPS now targeting production.
- Deployment to GCP (Cloud Run, Firebase Hosting, Cloud SQL, Secret Manager, Artifact Registry, Cloud Scheduler): **COMPLETED.**

8. Critical Next Steps / Remaining MVP Tasks (Updated Plan for Handover)

- **Task Ref: Original Task 13 & 12.2 - PO Export Feature (Backend & Frontend)**
    - **Status: PENDING**
    - **Goal:**
        - Backend: Create an endpoint (GET /api/exports/pos) that generates and returns an Excel (.xlsx) file summarizing Purchase Order details.
        - Frontend: Add a button/link on the Dashboard.jsx to trigger this PO export and download the file.
    - **Backend Action Items:**
        - Define Flask route in app.py.
        - Write SQL query
          (joins purchase_orders, po_line_items, suppliers, orders).
        - Use openpyxl (or similar) to generate .xlsx in memory.
        - Use Flask's send_file to return it.
        - Determine Excel sheet layout and required columns.
        - Handle data types and formatting (dates, currency) for Excel.
    - **Frontend Action Items:**
        - Add UI element ("Export POs" button) to Dashboard.jsx.

        - Button triggers GET request to /api/exports/pos.
        - Handle file download response.
        - Consider loading feedback and date range filters.
- **Task Ref: Original Task 15 - Comprehensive Testing**
    - **Status: PARTIALLY DONE (developer testing), FORMAL & FULL END-TO-END TESTING PENDING.**
    - **Action Items:**
        - **End-to-End Testing:** Test full flows (Dashboard -> Order Detail -> Process -> Verify DB/GCS/Email/BC updates) for various scenarios (direct HPE mapping, underscore fallback, no HPE mapping, manual SKU/desc overrides).
            - Specifically test with **live UPS Production API calls** for label generation, track a few, and understand any billing implications.
            - Test voiding UPS shipments if that functionality is critical.
        - **Edge Case Testing:** Test API errors, missing mappings, validation failures, international order handling (currently blocked from auto-processing, verify this block), network issues during multi-step processing.
        - **Data Validation:** Thoroughly test input validations on both frontend and backend, especially for the order processing payload.
        - **User Acceptance Testing (UAT):** Have the end-user(s) test the application thoroughly.
- **Task Ref: Original Task 6 (Postmark) - External Domain Sending Approval**
    - **Status: PENDING VERIFICATION** (Postmark internal domain sending works).
    - **Action Item:** Ensure the Postmark account is fully approved for sending emails to **external supplier domains** before full production rollout. This might require DNS changes (DKIM, SPF, Return-Path) for the sending domain used in EMAIL_SENDER_ADDRESS.
- **Task: Frontend - Eloquia Font Issue**

- **Status: PENDING (Not rendering on deployed site)**
- **Action Items:**

1. Re-verify the font file paths in the *source* App.css @font-face rules relative to the location of App.css and the font files (src/assets/fonts/ or public/fonts/).

2. Confirm font files are present in the dist folder after npm run build AND that the URLs in the *generated/deployed CSS* point to them correctly.

3. Critically examine the browser's Network Tab (filtered by "Font", with cache disabled, on hard refresh) on the deployed site to see if requests for the font files are being made and if they succeed (200 OK) or fail (404 Not Found).

4. If loading with 200 OK, use browser DevTools "Computed Styles" to see why 'Eloquia' isn't the active font-family (could be a typo in font-family: 'Eloquia' application or a more specific CSS rule overriding it).

5. Consider moving fonts to the public/fonts/ directory and updating @font-face URLs in App.css to be root-relative (e.g., url('/fonts/eloquia-extra-light.woff2')) as a more robust method with Vite.

- **Task: Cloudflare DNS for g1po.com**
  - **Status: PENDING**
  - **Goal:** Make the application accessible via g1po.com.
  - **Action Items (using Firebase Hosting for Frontend):**

1. In Firebase Hosting console, "Add custom domain" for g1po.com.

2. Firebase will provide DNS records (A records, possibly a TXT record).

3. In Cloudflare DNS settings for g1po.com, add/update these records. Set A records to "DNS Only" (grey cloud) initially for Firebase verification.

4. Wait for Firebase to verify and provision SSL.

5. Once https://g1po.com works, change A records in Cloudflare to "Proxied" (orange cloud).

6. (Optional but Recommended) Set up api.g1po.com:

    - Add CNAME api in Cloudflare pointing to the Cloud Run backend URL (g1-po-app-backend-service-....run.app), "Proxied".

        - Map api.g1po.com as a custom domain in the Cloud Run service settings in GCP.
        - Update VITE_API_BASE_URL in frontend's .env.production to https://api.g1po.com/api, rebuild, and redeploy frontend.
    - Alternatively, for a temporary solution, set up URL Forwarding in Cloudflare Page Rules from g1po.com to the https://g1-po-app-77790.web.app URL.
- **Task: Security Hardening & Best Practices**
    - **Status: PENDING**
    - **Action Items:**
        - **Cloud Run Authentication:** Transition from --allow-unauthenticated. Secure the /ingest_orders endpoint (if called by Cloud Scheduler) and other API endpoints. Use OIDC tokens for service-to-service authentication (e.g., Cloud Scheduler calling Cloud Run).
        - **IAM Least Privilege:** Create dedicated IAM service accounts for Cloud Run and Cloud Scheduler with only the minimum necessary roles, instead of using the default Compute Engine service account.
        - **Input Validation:** Review all backend API endpoints for robust input validation to prevent errors and potential security issues.
        - **Error Handling:** Enhance user-facing error messages on the frontend. Implement more structured logging on the backend (e.g., using Python's logging module with different levels).
- **Task Ref: Future Enhancement - Supplier and Product (HPE Mapping) Management UI**
    - **Status: PENDING (Placeholders exist)**
    - **Goal:** Develop frontend interfaces for CRUD operations on suppliers (suppliers table) and HPE part mappings (hpe_part_mappings table, potentially hpe_description_mappings as well).
    - **Action Items:** Design and implement these React components, connect them to existing backend CRUD API endpoints.
- **Task Ref: Recommended - Unit/Integration Tests (Backend)**

    - **Status: PENDING**
    - **Action Item:** Add automated tests for critical backend logic (HPE mapping, SKU fallback, PO number generation, status updates, API client interactions, database operations). Use pytest or Python's unittest.

9. Known Issues / Considerations (Updated)

- **UPS CIE Duplicate Tracking Numbers:** (Original issue, less relevant now with production but good to keep in mind if extensive test-mode is re-enabled). The test UPS (CIE) environment may return duplicate tracking numbers.
  Current shipping_service.py doesn't explicitly handle this.
- **CSS Variable Scope:** Global CSS variables are now primarily in App.css. Component-specific CSS files should *use* these global variables, not redefine them, for theme consistency.
- **Error Handling Granularity:** Current error handling is functional. Future enhancements could include more granular error catching/reporting on both backend and frontend.
- **International Orders:** Currently blocked from automated processing by process_order route. Future development needed for customs documentation, different shipping logic, etc.
- **Scalability:** The current synchronous design of process_order is suitable for moderate load. For very high volume, consider transitioning long-running tasks (PDF generation, external API calls) to an asynchronous task queue (e.g., Cloud Tasks, Celery).
- **Frontend State Management for "Order Details Not Found":** The brief flash of this message in OrderDetail.jsx was mitigated but might need further refinement of loading states if it persists as an issue.
- **Security (Secrets):** All API keys and sensitive credentials are managed via Google Cloud Secret Manager for Cloud Run. Local .env files are in .gitignore. Input validation on API endpoints should be continually reviewed.
- **UPS Production Costs:** Generating labels via the production UPS API will incur real costs. Test cautiously. Understand voiding procedures.
- **Postmark Deliverability:** Ensure sending domain for Postmark is properly authenticated (SPF, DKIM, DMARC, Custom Return-Path) to maximize deliverability to supplier emails.

10. Environment Setup Guide (Recap for New Developer)

- **Backend (Python/Flask):**
    1. Clone repository.
    2. Navigate to backend root (order-processing-app).
    3. Ensure Python 3.9+ is installed.
    4. Create a Python virtual environment: python -m venv venv
    5. Activate: .\venv\Scripts\Activate.ps1 (Windows PowerShell, after Set-ExecutionPolicy -Scope Process RemoteSigned) or source venv/bin/activate (Linux/macOS).
    6. Install dependencies: pip install -r requirements.txt.
    7. Create .env file in backend root (copy from an example or provided template, **DO NOT COMMIT ACTUAL SECRETS**). Populate with:
        - Database credentials (for local dev, often connect directly or via local Cloud SQL Proxy).
        - BigCommerce API credentials.
        - UPS API credentials (Client ID, Secret, Account #, Username/Password if still used by OAuth flow locally). Set UPS_API_ENVIRONMENT=test for local testing against CIE.
        - Postmark Server API Token, Sender/BCC addresses.
        - GCS Bucket Name (and set up local ADC or service account key for GCS access).
        - Ship From address details.
        - Other config like BC_PROCESSING_STATUS_ID.
    8. **For Local DB (Cloud SQL Proxy):**
        - Install gcloud CLI and authenticate (gcloud init, gcloud auth application-default login).
        - Download/run Cloud SQL Auth Proxy executable, pointing to the production Cloud SQL instance connection name.

        - Configure .env DB variables for proxy connection (host 127.0.0.1, port 5432, correct user/pass/db_name).
    9. Run Flask app locally: flask run (or python app.py).

- **Frontend (React/Vite):**
    1. Navigate to frontend root (order-processing-app-frontend).
    2. Ensure Node.js (LTS version) and npm are installed.
    3. Install dependencies: npm install.
    4. Create .env.development (or use .env) in frontend root for local API proxy: VITE_API_BASE_URL=/api
    5. Ensure vite.config.js has proxy setup for /api pointing to local backend (e.g., http://localhost:8080).
    6. Run Vite dev server: npm run dev. Access at http://localhost:5173 (or as per terminal output).
    7. For production builds that target the live backend, the .env.production file with VITE_API_BASE_URL=https://<cloud-run-url>/api is used by npm run build.

11. Deployment Process (Recap for Re-deployments)

- **Backend (to Cloud Run):**
    1. Make code changes in order-processing-app.
    2. Update requirements.txt if new Python packages are added.
    3. Rebuild Docker image: docker build -t <tag> .
    4. Push Docker image to Artifact Registry: docker push <tag>
    5. Redeploy Cloud Run service: gcloud run deploy <service-name> --image <tag> --update-env-vars=... --update-secrets=... ... (include all necessary flags).
- **Frontend (to Firebase Hosting):**
    1. Make code changes in order-processing-app-frontend.
    2. Update .env.production if VITE_API_BASE_URL changes.

3. Build for production: npm run build.
4. Deploy to Firebase: firebase deploy --only hosting.

12. Conclusion for Handover

Significant progress has been made on the G1 PO App. The core backend processing workflow is robust, deployed, and integrated with production UPS services. Key API integrations are functional. The frontend provides essential viewing, filtering, and interaction capabilities and is also deployed. The primary remaining MVP task is the PO Export feature. Following that, comprehensive testing, security hardening, and addressing the font rendering issue are key priorities. This document provides the context, current status, and a refined roadmap for the incoming developer to successfully complete the MVP, connect the custom domain, and move the application towards full production readiness.

This report should give the new developer a very solid understanding of where the project stands and what needs to be done. Good luck to them!