G1 PO App - Comprehensive Handover Report & Next Steps

Date: May 14, 2025 (Reflecting current progress and chat session ending May 14, 2025 CDT)

Prepared For: Incoming Developer

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

Version: 3.0 (Supersedes report dated May 13, 2025, version 2.1)

1. Introduction

This document provides an updated and comprehensive overview and handover for the "G1 PO App" project. It consolidates information from previous handover reports with all subsequent development progress. This includes the implementation and extensive debugging of the daily IIF file generation for QuickBooks Desktop 2020, critical updates to backend services (`app.py`, `shipping_service.py`), the successful setup of automated task triggering via Google Cloud Scheduler, and most recently, the integration of **Firebase Authentication (Google Sign-In)** for securing the application. The goal remains to provide a clear and detailed guide for the incoming developer to understand the project's current state, architecture, codebase, deployment, security model, and the remaining tasks to achieve the Minimum Viable Product (MVP) and future enhancements.

2. Project Overview (Recap)

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, with secure access for authorized users.

Core Functionality (Original Intent & Current Implementation):

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

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

* Allow authorized users to trigger processing for selected orders.

* Generate Purchase Orders (PDFs) using specific HPE Option PNs mapped from original BigCommerce SKUs.

* Generate Packing Slips (PDFs).

* Generate UPS Shipping Labels (PDFs) via UPS API (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 (`orders` and `purchase_orders` tables).

* Provide a web interface (React/Vite) for viewing/filtering orders, viewing details, interactively preparing POs, and triggering processing – **now secured with user authentication**.

* **COMPLETED & VERIFIED:** Generate a daily IIF (Intuit Interchange Format) file for Purchase Orders processed on the previous day, suitable for import into QuickBooks Desktop 2020. This IIF file is emailed to a designated address.

* **NEW:** User authentication via Firebase (Google Sign-In).

* **NEW (Basic):** Authorization mechanism using Firebase Custom Claims to restrict app access to approved Google accounts.

* (Future) Manage suppliers and product mappings more extensively via the UI.

* (Future) More robust admin interface for managing user approvals.

3. Current Status (As of May 14, 2025)

The application is largely functional, with core backend processing deployed, IIF generation operational, and a foundational user authentication and authorization system implemented.

3.1. Backend (Python/Flask - `app.py` and service modules)

* **Deployment:** Dockerized and deployed to Google Cloud Run.
  * Service Name: `g1-po-app-backend-service`
  * Region: `us-central1`
  * URL: `https://g1-po-app-backend-service-992027428168.us-central1.run.app` (Note: Cloud Run URL might change on redeployments if not mapped to a custom domain).
* **Database:** Connected to a PostgreSQL instance on Google Cloud SQL.
  * Instance Connection Name: `order-processing-app-458900:us-central1:order-app-db`
  * Connection managed via Cloud SQL Auth Proxy sidecar in Cloud Run (or direct connection if configured).
* **Secrets Management:** All sensitive credentials managed via Google Cloud Secret Manager.
* **Key API Endpoints:**
  * `/api/ingest_orders` (POST): Functional. Fetches orders from BigCommerce. Logic updated to preserve manually set statuses like "Pending", "Completed Offline", "RFQ Sent" rather than reverting them.
  * `/api/orders`, `/api/orders/<id>` (GET): Functional.
  * `/api/orders/<id>/process` (POST): Core workflow functional.
  * `/api/orders/<id>/status` (POST): Functional for manual status updates (e.g., to "Pending", "Completed Offline").
  * `/api/orders/status-counts` (GET): **NEW** endpoint to provide counts of orders by status for the frontend dashboard filter.

  * `/api/suppliers/**`, `/api/products/**` (GET, POST, PUT, DELETE): CRUD endpoints for suppliers and product mappings are functional.

  * `/api/lookup/description/**` (GET): Functional.

  * `/api/tasks/trigger-daily-iif` (POST): Functional, triggered by Cloud Scheduler.

* **Authentication & Authorization (NEW):**

  * **Firebase Admin SDK:** Initialized in `app.py` to verify Firebase ID tokens.

  * **`@verify_firebase_token` Decorator:** Applied to all data-accessing and action-performing API routes (except the IIF task trigger). This decorator validates the `Bearer` token from the `Authorization` header.

  * **Custom Claim Check:** The `verify_firebase_token` decorator now checks for a custom claim `isApproved: true` on the decoded Firebase ID token. Users without this claim (or with it set to false) will receive a 403 Forbidden error, even if their Google Sign-In was successful.

* **CORS:** Configured in `app.py` to allow requests from the Firebase Hosting frontend URL and `http://localhost:5173` for local development.

* **Error Handling:** UPS label generation error ("Missing or invalid ship to state province code") was addressed by implementing proper state code mapping in `shipping_service.py`.

3.2. Frontend (React/Vite - `order-processing-app-frontend`)

* **Deployment:** Deployed to Firebase Hosting.

  * URL: `https://g1-po-app-77790.web.app`

* **Authentication & Authorization (NEW):**

  * **Firebase SDK:** Integrated for client-side authentication. `src/firebase.js` (or `firebaseConfig.js`) holds the project configuration.

  * **Google Sign-In:** Implemented as the primary authentication method.

  * **`AuthContext.jsx`:** Created to manage global authentication state (`currentUser`, `loading` state for auth resolution, `signInWithGoogle`, `logout` functions).

* **`Login.jsx`:** New component providing the "Sign In with Google" button (using a custom image `src/assets/sign-in-button.png`). Redirects if user is already logged in.

* **`ProtectedRoute.jsx`:** New component to guard routes. Redirects unauthenticated users to `/login` and shows a loading message while auth state is being determined.

* **`App.jsx`:** Updated to include `AuthProvider` at the root, and routes are now structured with `ProtectedRoute` for all authenticated sections. Navigation bar dynamically changes based on auth state.

* **Component Updates for Auth:**

  * `Dashboard.jsx`, `OrderDetail.jsx`, `SupplierList.jsx`, `SupplierForm.jsx`, `EditSupplierForm.jsx`, `ProductMappingList.jsx`, `ProductMappingForm.jsx`, `EditProductMappingForm.jsx` have been updated (or need to be fully updated) to:

    * Use `useAuth()` to get `currentUser`.

    * Send the Firebase ID token (`Authorization: Bearer <token>`) with all API requests to the backend.

    * Handle UI changes based on auth state (e.g., disabling buttons, showing login prompts if data cannot be fetched).

    * Handle 401/403 errors from the backend.

* **UI Changes:**

  * Dashboard: Filter dropdown now has a specific order for statuses ("New", "RFQ Sent", "Pending", "International", "Processed", "Completed Offline") and displays counts. An asterisk indicates if "Pending" or "International" orders exist. "All Orders" option removed.

  * OrderDetail:

    * "Mark as Completed Offline" button added for "International" or "Pending" orders (changes status to "Completed Offline" - a new status).

    * "Process Manually (Set to Pending)" action (now a link) added for "New" orders.

    * "PROCESS ORDER" button is now hidden for "Pending" and "International" statuses.

  * Styling for "Completed Offline" status badge updated for better visibility (black background, white text).

* **Font Issue:** Eloquia font rendering issue on the deployed Firebase site (as noted in previous report) likely still pending.

3.3. Database (PostgreSQL on Cloud SQL)

* Core tables (`orders`, `order_line_items`, `purchase_orders`, `po_line_items`, `suppliers`, `hpe_part_mappings`, `hpe_description_mappings`, `products`, `qb_product_mapping`, `shipments`) are defined and functional.

* **NEW Status:** `Completed Offline` added as a valid order status.

* `orders.status` is actively used and updated by both automated processing and manual actions.

3.4. Integrations

* **UPS API:** Functional (state code mapping issue resolved).

* **BigCommerce API:** Functional.

* **Postmark API:** Functional for all email types.

* **Google Cloud Storage (GCS):** Functional for document uploads.

* **Firebase Authentication (NEW):** Integrated for user sign-in (Google) and token verification.

3.5. Scheduling (IIF Generation)

* Google Cloud Scheduler: Job configured to call `/api/tasks/trigger-daily-iif` via HTTP POST.

* Authentication: OIDC token using a dedicated service account with "Cloud Run Invoker" role. This remains appropriate for server-to-server calls and should *not* use the Firebase user authentication.

4. Technology Stack (Recap & Additions)

* **Backend:** Python 3.9+, Flask, SQLAlchemy, pg8000, requests, google-cloud-sql-connector, google-cloud-storage, reportlab, Pillow, Flask-CORS, postmarker, python-dotenv, gunicorn, **firebase-admin**.

* **Frontend:** React, Vite, react-router-dom, axios (if still used, or `fetch` API), **firebase** (client SDK).

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

* **Cloud Platform (GCP):** Cloud Run, Cloud SQL, Artifact Registry, Secret Manager, GCS, Cloud Scheduler.

* **Firebase:** Firebase Hosting (for frontend), Firebase Authentication (for user auth).

* **APIs:** UPS, BigCommerce, Postmark.

5. Codebase Structure (Key Files & Recent Changes)

5.1. Backend (`order-processing-app` directory)

* **`app.py`:**

  * Firebase Admin SDK initialized.

  * `@verify_firebase_token` decorator created and applied to relevant API routes. This decorator checks for a valid Firebase ID Token and the `isApproved: true` custom claim.

  * `ingest_orders` logic updated to preserve manually set statuses like "Pending", "Completed Offline", "RFQ Sent".

  * `/api/orders/status-counts` endpoint added.

  * `allowed_statuses` list in `update_order_status` endpoint updated to include "Completed Offline".

* **`requirements.txt`:**

  * `firebase-admin` added as a dependency. (Ensure specific version conflicts, especially with `google-api-core`, are resolved by allowing pip to manage sub-dependencies or using compatible versions).

* **`shipping_service.py`:** State code mapping for UPS labels fixed.

* **`setAdminClaim.cjs` (External Script):**

  * A Node.js script (not part of the deployed application) created for manually setting custom claims (like `isApproved: true`) on Firebase users.

  * Requires Firebase Admin SDK and a service account key JSON file.

  * **Crucially, this script and the service account key must be stored securely and NOT committed to any Git repository.**

5.2. Frontend (`order-processing-app-frontend` directory)

* **`src/firebase.js` (or `firebaseConfig.js`):**

  * Contains the Firebase project configuration (apiKey, authDomain, projectId, etc.).

  * Initializes Firebase app and exports `auth` service.

* **`src/contexts/AuthContext.jsx` (NEW):**

  * Provides global authentication state (`currentUser`, `loading`, `signInWithGoogle`, `logout`).

  * Uses `onAuthStateChanged` to listen for auth state changes.

* **`src/components/Login.jsx` (NEW):**

  * Provides the UI for Google Sign-In (using a custom image button `src/assets/sign-in-button.png`).

  * Handles the sign-in process via `AuthContext`.

* **`src/components/ProtectedRoute.jsx` (NEW):**

  * A wrapper component that checks authentication state.

  * Redirects to `/login` if the user is not authenticated.

  * Shows a loading message while auth state is being resolved.

* **`src/App.jsx`:**

  * Wrapped with `<BrowserRouter>` in `main.jsx`.

  * Now wraps its main content (`AppContent`) with `<AuthProvider>`.

  * Routing structure updated to use `ProtectedRoute` for all authenticated routes.

  * Navigation bar dynamically updates based on `currentUser` (shows Login/Logout links).

* **Authenticated Components (e.g., `Dashboard.jsx`, `OrderDetail.jsx`, `SupplierList.jsx`, `SupplierForm.jsx`, `EditSupplierForm.jsx`, `ProductMappingForm.jsx`, `EditProductMappingForm.jsx`):**

  * Import and use `useAuth()` to get `currentUser` and `authLoading`.

  * API calls (`fetch`) are updated to include `Authorization: Bearer <ID_TOKEN>` header.

  * Data fetching and UI elements (buttons, forms) are made conditional on `currentUser` and `authLoading` states.

  * Error handling for 401/403 responses from the backend needs to be robust.

6. API Integrations & Credentials

* **Backend Secrets:** Managed in Google Cloud Secret Manager for deployed services (DB credentials, BigCommerce tokens, UPS keys, Postmark token, etc.).

* **Local Development (`.env` files):**

  * Backend: `.env` file in `order-processing-app` for all backend secrets.

    * **NEW:** `GOOGLE_APPLICATION_CREDENTIALS` environment variable should point to the Firebase Admin SDK service account key JSON file for local backend development if not relying on `gcloud auth application-default login` for Admin SDK.

  * Frontend: `.env.development` and `.env.production` in `order-processing-app-frontend` for `VITE_API_BASE_URL`.

* **Firebase Project Setup:**

  * Web app registered in the Firebase project.

  * Firebase Authentication enabled, with Google Sign-In as a provider.

* **Service Account Key (Firebase Admin):**

* Generated from Firebase Console.

* **EXTREMELY SENSITIVE.** Used by `setAdminClaim.cjs` and potentially by the local backend for Firebase Admin SDK initialization.

* **MUST NOT BE COMMITTED TO GIT.** Store securely.

* **GitHub Secret Scanning:**

* Alerts for the Firebase Web `apiKey` in `frontend/src/firebase.js` can be dismissed as "Key is used in client-side code" or "Not a secret," as this key is designed to be public. Security relies on Firebase Security Rules and backend ID token verification.

* Any other true secrets flagged by GitHub must be addressed immediately (revoke, remove from history, use secret management).

7. Authentication & Authorization Flow (NEW)

1. **User Navigates to App:**

* If the user visits a protected route (e.g., `/dashboard`) and is not authenticated, `ProtectedRoute.jsx` redirects them to `/login`.

* If they are already authenticated (Firebase session persists), `AuthContext` sets `currentUser`, and `ProtectedRoute.jsx` allows access.

2. **Login (`Login.jsx`):**

* User clicks the "Sign In with Google" button (custom image).

* `signInWithGoogle()` from `AuthContext` is called, which triggers `signInWithPopup(auth, googleProvider)`.

* Firebase handles the Google Sign-In flow in a popup (Google's UI, including account chooser if multiple accounts are signed into the browser).

3. **Token Issuance:**

* Upon successful Google Sign-In, Firebase Authentication mints an ID Token (JWT) for the user.

* `onAuthStateChanged` in `AuthContext.jsx` fires, setting `currentUser` with the user object.

4. **API Requests from Frontend:**

   * For any API call to the backend (e.g., fetching orders, submitting a form):

      * The React component gets `currentUser` from `useAuth()`.

      * It calls `await currentUser.getIdToken(true)` to get a fresh ID Token.

      * The ID Token is included in the `Authorization` header of the `fetch` request: `Authorization: Bearer <ID_TOKEN>`.

5. **Backend Token Verification (`app.py` - `@verify_firebase_token` decorator):**

   * The Flask backend receives the request.

   * The `@verify_firebase_token` decorator extracts the `Bearer` token.

   * It uses `firebase_auth.verify_id_token(id_token)` from the Firebase Admin SDK to:

      * Verify the token's signature and integrity.

      * Check if it's expired.

      * Ensure it was issued by your Firebase project.

   * If verification fails, a 401 Unauthorized error is returned.

6. **Backend Authorization (Custom Claim Check):**

   * If the ID token is valid, the `verify_firebase_token` decorator then inspects the `decoded_token` for a custom claim: `isApproved: true`.

   * If this claim is missing or not true, a 403 Forbidden error is returned, even though the user is legitimately authenticated with Google.

   * If the claim is present and true, the request proceeds to the route handler. User information (UID, email) is available via `flask.g.user_uid` / `g.user_email`.

7. **User Approval (Setting Custom Claims):**

   * Currently, this is a manual process using the `setAdminClaim.cjs` script.

   * **Process:**

      1. A new user signs into the app for the first time (this creates their record in Firebase Auth).

      2. An administrator finds the user's UID in the Firebase Authentication console (Users tab).

3. The administrator runs `setAdminClaim.cjs` locally, providing this UID and the service account key, to set the `{ isApproved: true }` claim.

4. The user might need to sign out and sign back in, or their ID token needs to refresh for the backend to see the new claim on subsequent requests. (The frontend can force a token refresh with `getIdToken(true)`).

8. Key Features Implemented / Progress Details (Summary)

* Core order processing workflow (PO, Packing Slip, UPS Label generation, GCS upload, supplier email, BigCommerce update): Largely functional, with recent bug fixes.

* Daily IIF Generation for QuickBooks Desktop POs: COMPLETED & VERIFIED.

* Automated triggering of IIF generation via Cloud Scheduler: SETUP & VERIFIED.

* **NEW:** User Authentication via Firebase (Google Sign-In) implemented on frontend.

* **NEW:** Backend API protection: All relevant API endpoints now require a valid Firebase ID Token.

* **NEW:** Basic Authorization: Access restricted to users with an `isApproved: true` custom Firebase claim.

* Frontend UI updates for dashboard filters and OrderDetail page actions.

* Resolution of critical backend bugs (UPS state code, `ingest_orders` status preservation).

* Resolution of frontend build errors related to JSX and module resolution.

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

This section outlines tasks to complete the MVP and further enhance the application.

9.1. Finalize Authentication & Authorization

1. **Thorough Testing:**

   * **Action:** Test all login, logout, and session persistence scenarios.

   * **Action:** Verify that all protected routes correctly redirect unauthenticated users.

   * **Action:** Test access with an **approved** Google account – ensure all functionalities work.

   * **Action:** Test access with a **non-approved** Google account (authenticated by Google but lacking the `isApproved` claim) – ensure backend returns 403 and frontend handles this gracefully (e.g., "Not Authorized" message, no access to data/actions).

   * **Action:** Test how the application behaves if an ID token expires mid-session (backend should return 401, frontend should ideally prompt for re-login or redirect).

2. **Robust Frontend Error Handling for 401/403:**

   * **Action:** In all API call sites (e.g., `Dashboard.jsx`, `OrderDetail.jsx`, forms), enhance error handling to specifically detect 401/403 responses.

   * **Action:** Upon receiving a 401/403, display a clear message to the user. Consider automatically logging the user out (`await logout()`) and redirecting to `/login` (`navigate('/login')`) for a smoother UX, especially for 401s.

3. **Admin User Approval Process:**

   * **Current:** Manual `setAdminClaim.cjs` script.

   * **Action (MVP):** Provide clear, step-by-step documentation for the administrator on how to use `setAdminClaim.cjs` securely (including managing the service account key).

   * **Action (Future Enhancement):** Design and implement a simple, secure admin interface within the G1 PO App itself (e.g., a new protected route `/admin/users`) where an admin user (identified by another custom claim like `isAdmin: true`) can view users and set their `isApproved` claim. This would likely involve a new backend endpoint callable only by admins to trigger a Cloud Function that sets the claim.

4. **Review Cloud Run Service Account IAM Permissions:**

   * **Action:** Ensure the service account used by the Cloud Run backend instance has only the necessary permissions for Firebase Admin SDK (to verify tokens) and other GCP services (Cloud SQL, GCS, Secret Manager). Follow the principle of least privilege.

9.2. Complete Frontend Authentication Integration

* **Action:** Audit all React components that make API calls. Ensure every single one:
  * Uses `useAuth()` to get `currentUser`.
  * Sends the ID token in the `Authorization` header.
  * Handles loading states related to `authLoading`.
  * Disables interactive elements if `!currentUser`.
  * Gracefully handles API errors, especially 401/403.
  * (This includes `SupplierList.jsx`, `SupplierForm.jsx`, `EditSupplierForm.jsx`, `ProductMappingList.jsx`, `ProductMappingForm.jsx`, `EditProductMappingForm.jsx` if not already fully updated).

9.3. PO Export Feature (Excel - Backend & Frontend)

* **Status:** PENDING (from previous report).
* **Goal:**
  * **Backend:** Create GET `/api/exports/pos` endpoint to generate and return an Excel (.xlsx) file summarizing Purchase Order details.
  * **Frontend:** Add a button/link on `Dashboard.jsx` (perhaps with date range filters) to trigger this export.
* **Technical Details:**
  * Backend: Use Flask's `send_file` or `make_response`. Library: `openpyxl`.
  * SQL Query: Join `purchase_orders`, `po_line_items`, `suppliers`.
  * Frontend: API call, handle file download response (Blob, `URL.createObjectURL`).

9.4. Comprehensive End-to-End Testing

* **Status:** PARTIALLY DONE. FORMAL & FULL TESTING PENDING.

* **Action Items:**

  * Test full order processing flows with authenticated (and authorized) users.

  * Test all edge cases for API integrations (UPS, BigCommerce, Postmark).

  * Thoroughly test all frontend interactions, forms, and error displays across different user auth states.

  * User Acceptance Testing (UAT) with the end-user(s).

9.5. Data Integrity for QuickBooks Lists (TERMS and SHIPVIA - Future Enhancement)

* **Status:** Currently N/A (blanked in IIF).

* **Goal (If desired later):** Populate TERMS and SHIPVIA in IIF.

* **Action:** Requires ensuring data matches QuickBooks lists exactly. May involve UI changes for data entry or backend mapping.

9.6. Postmark - External Domain Sending Approval

* **Status:** PENDING VERIFICATION.

* **Action Item:** Ensure Postmark account is fully approved for sending to external supplier domains (DNS changes: DKIM, SPF, Custom Return-Path for sending domain).

9.7. Frontend - Eloquia Font Issue

* **Status:** PENDING.

* **Action Items:** Investigate and resolve font rendering issue on deployed Firebase site.

9.8. Cloudflare DNS for `g1po.com` & `api.g1po.com`

* **Status:** PENDING.

* **Action Items:**

  * Frontend: Add `g1po.com` as custom domain in Firebase Hosting, update DNS in Cloudflare.

  * Backend (Optional but Recommended): Set up `api.g1po.com` pointing to Cloud Run service, map custom domain in Cloud Run, update `VITE_API_BASE_URL` in frontend.

9.9. Further Security Hardening & Best Practices

* **Cloud Run Ingress:**

  * Currently uses `--allow-unauthenticated` at the Cloud Run service level, with authentication handled at the application level by `verify_firebase_token`.

  * **Action:** Evaluate if this is sufficient for production or if GCP-level authentication (e.g., requiring an OIDC token for all calls, even from the frontend, potentially via IAP - Identity-Aware Proxy) should be layered on top. Application-level auth is a strong start.

* **Input Validation:**

  * **Action:** Re-emphasize and conduct a thorough review of all backend API endpoints for robust input validation on all incoming data (query parameters, JSON bodies).

* **Error Handling & Logging:**

  * **Action:** Enhance user-facing error messages on the frontend for clarity.

  * **Action:** Implement more structured and detailed logging on the backend (Python's `logging` module, levels, effective use of Google Cloud Logging).

9.10. Supplier and Product (HPE Mapping & QB Mapping) Management UI

* **Status:** PENDING (Basic forms might exist).

* **Goal:** Develop user-friendly frontend interfaces for CRUD operations on `suppliers`, `hpe_part_mappings`, `hpe_description_mappings`, `qb_product_mapping` tables.

* **Action Items:** Design and implement React components, connect to backend CRUD APIs (ensuring these APIs are also protected by `@verify_firebase_token`).

9.11. Unit/Integration Tests (Backend & Frontend)

* **Status:** PENDING.

* **Action Item:** Add automated tests (e.g., `pytest` for backend, Jest/React Testing Library for frontend) for critical logic.

10. Known Issues / Considerations (Updated)

* **IIF Import Sensitivities:** QuickBooks Desktop IIF import is very sensitive.

* **UPS Production Costs:** Real costs will be incurred with production UPS API.

* **Postmark Deliverability:** DNS setup is crucial.

* **Scalability of Synchronous Processing:** Consider asynchronous task queues for high volumes.

* **Environment Management:**

  * Ensure robust separation for local dev, staging (if any), and production.

  * Frontend `VITE_API_BASE_URL` needs to be correct for each environment.

  * Backend `.env` / Secret Manager for credentials.

  * **NEW:** Firebase project configuration in `frontend/src/firebase.js` is client-side and typically the same for dev/prod unless using different Firebase projects per environment (which adds complexity but is possible).

* **Custom Claim Management (NEW):** The current `setAdminClaim.cjs` script is a manual, developer-run tool. A more user-friendly and secure system for managing user approvals (`isApproved` claim) is needed for production administration by non-developers.

* **Firebase Quotas (NEW):** Monitor Firebase Authentication quotas (e.g., sign-ins, token creations) in the Firebase and Google Cloud Consoles, especially "Identity Toolkit API" quotas. The `auth/quota-exceeded` error was encountered during development and indicates that frontend API call patterns should be efficient to avoid hitting limits.

* **Service Account Key Security:** The Firebase Admin SDK service account key (`.json` file) is highly sensitive. It must **never** be committed to Git or exposed publicly. For local backend development, it can be referenced via the `GOOGLE_APPLICATION_CREDENTIALS` environment variable. For Cloud Run, the service's runtime service account with appropriate IAM permissions is preferred over embedding a key file. The `setAdminClaim.cjs` script requires local access to this key.

11. Environment Setup Guide (Recap & Update for New Developer)

11.1. Backend (Python/Flask - `order-processing-app` directory)

1. Clone repository.

2. Navigate to backend root.

3. Ensure Python 3.9+ is installed.

4. Create/activate Python virtual environment (e.g., `python -m venv .venv` then `source .venv/bin/activate` or `.\.venv\Scripts\activate`).

5. Install dependencies: `pip install -r requirements.txt`. (Ensure `firebase-admin` is in `requirements.txt`).

6. Create `.env` file in backend root (from template, **DO NOT COMMIT SECRETS**). Populate with:

   * Database credentials (for local dev, often connect directly or via local Cloud SQL Proxy).

   * BigCommerce API credentials.

   * UPS API credentials.

   * Postmark Server API Token, Sender/BCC addresses.

   * GCS Bucket Name.

* **`GOOGLE_APPLICATION_CREDENTIALS` (for local dev):** Path to your downloaded Firebase Admin SDK service account key JSON file (e.g., `C:/path/to/your-service-account-key.json`). This allows the local Flask app to initialize `firebase-admin`.

* Ship From address details.

* Other config like `BC_PROCESSING_STATUS_ID`, `QUICKBOOKS_EMAIL_RECIPIENT`, etc.

7. For Local DB (Cloud SQL Proxy):

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

* Download/run Cloud SQL Auth Proxy, pointing to the production Cloud SQL instance.

* Configure `.env` DB variables for proxy connection (host `127.0.0.1`, port `5432`).

8. Run Flask app locally: `flask run` or `python app.py`.

11.2. Frontend (React/Vite - `order-processing-app-frontend` directory)

1. Clone repository.

2. Navigate to frontend root.

3. Ensure Node.js (LTS version) and npm (or yarn) are installed.

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

5. **Firebase Setup:**

* Create `src/firebase.js` (or `firebaseConfig.js`).

* Go to your Firebase project in the Firebase Console.

* Project settings (gear icon) > Your apps.

* If no web app is registered, click "Add app" (`</>`), give it a nickname (e.g., "G1 PO Web App"), and register it. **You do not need to enable Firebase Hosting again through this flow if it's already set up.**

* Copy the `firebaseConfig` object (containing `apiKey`, `authDomain`, `projectId`, `storageBucket`, `messagingSenderId`, `appId`) from the "SDK setup and configuration" section into your `src/firebase.js` file.

* Ensure `src/firebase.js` initializes Firebase and exports the `auth` service:

```javascript
import { initializeApp } from "firebase/app";
import { getAuth } from "firebase/auth";

const firebaseConfig = { /* ...pasted config... */ };
const app = initializeApp(firebaseConfig);
export const auth = getAuth(app);
```

6. Create `.env.development` in frontend root for local API URL:

* `VITE_API_BASE_URL=http://localhost:8080/api` (or your local backend URL).

7. Create `.env.production` for deployed API URL:

* `VITE_API_BASE_URL=https://your-cloud-run-backend-url/api`

8. Ensure `vite.config.js` has proxy setup for `/api` if you are proxying local dev requests (optional, direct URL in `.env.development` is also fine).

9. Run Vite dev server: `npm run dev`.

12. Deployment Process (Recap & Update)

12.1. Backend (to Cloud Run)

1. Make code changes in `order-processing-app`.

2. Update `requirements.txt` if new Python packages (like `firebase-admin`) are added.

3. Rebuild Docker image: `docker build -t gcr.io/YOUR_PROJECT_ID/g1-po-app-backend-service:YOUR_TAG .`

4. Push Docker image: `docker push gcr.io/YOUR_PROJECT_ID/g1-po-app-backend-service:YOUR_TAG`

5. Redeploy Cloud Run service:

   ```bash

   gcloud run deploy g1-po-app-backend-service \

     --image gcr.io/YOUR_PROJECT_ID/g1-po-app-backend-service:YOUR_TAG \

     --region YOUR_REGION \

     --allow-unauthenticated \ # Application-level auth is now handled by verify_firebase_token

     --set-env-vars "GOOGLE_APPLICATION_CREDENTIALS=" \ # For Cloud Run, rely on service account identity for Admin SDK

     # ... include all other necessary flags for env vars from Secret Manager, Cloud SQL instance, etc.

   ```

   * **Note on `GOOGLE_APPLICATION_CREDENTIALS` for Cloud Run:** It's generally better to *not* set this environment variable directly in Cloud Run if your service account has the correct IAM permissions. The Firebase Admin SDK will auto-discover credentials in GCP environments. If you were using a key file *in* Cloud Run (not recommended), you'd set it. For now, relying on the service account is cleaner.

12.2. Frontend (to Firebase Hosting)

1. Make code changes in `order-processing-app-frontend`.

2. Ensure `VITE_API_BASE_URL` in `.env.production` points to the correct deployed backend URL.

3. Build for production: `npm run build`.

4. Deploy to Firebase: `firebase deploy --only hosting` (or `firebase deploy --only hosting:g1-po-app-77790` to be explicit).

13. Conclusion for Handover

Significant progress has been made. The G1 PO App now has a functional core backend, IIF generation, and a foundational user authentication (Google Sign-In) and authorization (custom claims) system. Frontend components have been progressively updated to integrate with this security model.

The immediate next steps involve completing and thoroughly testing the authentication/authorization flow across all components, implementing a more robust admin process for user approvals, and then tackling the remaining MVP features like the Excel PO export. This document, along with the codebase and existing cloud infrastructure, should provide the incoming developer with a solid and detailed foundation to successfully complete the MVP and move the application towards full production readiness.