# DIG Assistant — End-to-End Build Guide (Image Search + React)

This guide shows every step (in order) to build your **Drawing Insight Generation (DIG) Assistant** with:

* **Python** pipeline for image ingestion + vector search (CLIP).
* **Astra DB (Cassandra + Vector)** for storage and search.
* **Node.js (Express)** API for the React app.
* **React** front-end inventory search page.

## 0) What you’ll build (flow)

**User (browser)** ⇄ **React UI** ⇄ **Express API** ⇄ **Astra DB (tables: items + images + vectors)**

* Search by **FG Item** to list batches & basic info.
* Check whether an **image/drawing exists** for that item.
* If not found, **ingest** a new image and **embed** it for future search.
* (Optional) Search by **text** → returns **nearest images** by vector similarity.

## 1) Prerequisites

* Astra DB database (Vector enabled).
* Download **Secure Connect Bundle** (secure-connect-<db>.zip).
* **Astra Application Token** (starts with AstraCS:).
* Python 3.9+
* Node.js 18+

Create a working folder with three subfolders:

DIG-assistant/  
 backend/ # Node.js Express API  
 frontend/ # React app  
 python\_pipeline/ # Python ingestion & search

## 2) Create Keyspace & Tables (CQL)

Open **Astra CQL Console** and run:

-- 2.1: Items (inventory) table  
CREATE TABLE IF NOT EXISTS fg\_items (  
 fgitem TEXT PRIMARY KEY,  
 batch\_number TEXT,  
 quantity INT,  
 site TEXT,  
 modified\_date TIMESTAMP  
);  
  
-- 2.2: Item Images table (multiple images per item)  
-- Using CLIP embeddings (512 dims)  
CREATE TABLE IF NOT EXISTS fg\_item\_images (  
 fgitem TEXT,  
 image\_id TIMEUUID,  
 file\_name TEXT,  
 image\_bytes BLOB,  
 embedding VECTOR<float, 512>,  
 uploaded\_at TIMESTAMP,  
 PRIMARY KEY (fgitem, image\_id)  
) WITH CLUSTERING ORDER BY (image\_id DESC);

If you also store PDFs with text embeddings, create a separate table with VECTOR<float, 1536> and ingest via text embeddings.

## 3) Python pipeline (image ingestion + similarity search)

Navigate to python\_pipeline/.

### 3.1 Install dependencies

Create requirements.txt:

cassandra-driver  
sentence-transformers  
torch  
Pillow  
numpy  
python-dotenv

Install:

pip install -r requirements.txt

### 3.2 Environment variables

Create .env in python\_pipeline/:

ASTRA\_BUNDLE=../secure-connect-yourdb.zip  
ASTRA\_TOKEN=AstraCS:YOUR\_LONG\_TOKEN  
ASTRA\_KEYSPACE=your\_keyspace  
SIM\_THRESHOLD=0.90

### 3.3 Python code — ingest\_and\_search.py

import os  
import io  
import uuid  
import time  
import numpy as np  
from datetime import datetime  
from dotenv import load\_dotenv  
from PIL import Image  
  
from cassandra.cluster import Cluster  
from cassandra.auth import PlainTextAuthProvider  
from cassandra.query import dict\_factory  
  
from sentence\_transformers import SentenceTransformer  
  
load\_dotenv()  
  
ASTRA\_BUNDLE = os.getenv("ASTRA\_BUNDLE")  
ASTRA\_TOKEN = os.getenv("ASTRA\_TOKEN")  
KEYSPACE = os.getenv("ASTRA\_KEYSPACE")  
SIM\_THRESHOLD = float(os.getenv("SIM\_THRESHOLD", 0.90))  
  
# ---- Cassandra connection ----  
cloud\_config = {"secure\_connect\_bundle": ASTRA\_BUNDLE}  
# For Astra token auth, username='token', password='<AstraCS:...>'  
auth\_provider = PlainTextAuthProvider("token", ASTRA\_TOKEN)  
cluster = Cluster(cloud=cloud\_config, auth\_provider=auth\_provider)  
session = cluster.connect(KEYSPACE)  
session.row\_factory = dict\_factory  
  
# ---- Model (CLIP) ----  
# Produces 512-dim embeddings for images AND text (cross-modal retrieval)  
model = SentenceTransformer("clip-ViT-B-32")  
  
# ---- Helpers ----  
def read\_image\_bytes(path: str) -> bytes:  
 with open(path, "rb") as f:  
 return f.read()  
  
def embed\_image\_bytes(img\_bytes: bytes) -> np.ndarray:  
 img = Image.open(io.BytesIO(img\_bytes)).convert("RGB")  
 vec = model.encode([img], convert\_to\_numpy=True, normalize\_embeddings=True)[0]  
 return vec.astype(np.float32)  
  
def embed\_text(text: str) -> np.ndarray:  
 vec = model.encode([text], convert\_to\_numpy=True, normalize\_embeddings=True)[0]  
 return vec.astype(np.float32)  
  
# ---- Items API (basic demo inserts) ----  
def upsert\_item(fgitem: str, batch\_number: str, quantity: int, site: str):  
 session.execute(  
 """  
 INSERT INTO fg\_items (fgitem, batch\_number, quantity, site, modified\_date)  
 VALUES (%s, %s, %s, %s, toTimestamp(now()))  
 """,  
 (fgitem, batch\_number, quantity, site),  
 )  
  
# ---- Images API ----  
# 1) Upsert image for an item: if similar image already exists -> report FOUND; else insert  
  
def upsert\_image\_for\_item(fgitem: str, image\_path: str):  
 img\_bytes = read\_image\_bytes(image\_path)  
 emb = embed\_image\_bytes(img\_bytes)  
  
 # Fetch existing embeddings for the item  
 rows = session.execute(  
 "SELECT image\_id, file\_name, embedding FROM fg\_item\_images WHERE fgitem=%s",  
 (fgitem,),  
 )  
  
 best\_sim = -1.0  
 best\_id = None  
 for r in rows:  
 sim = float(np.dot(emb, np.array(r["embedding"], dtype=np.float32))) # unit-normalized vectors  
 if sim > best\_sim:  
 best\_sim = sim  
 best\_id = r["image\_id"]  
  
 if best\_sim >= SIM\_THRESHOLD:  
 print(f"✅ Image already exists for {fgitem} (similarity={best\_sim:.3f}, image\_id={best\_id})")  
 return "found", best\_id, best\_sim  
  
 # Insert new  
 image\_id = uuid.uuid1() # time-sortable  
 session.execute(  
 """  
 INSERT INTO fg\_item\_images (fgitem, image\_id, file\_name, image\_bytes, embedding, uploaded\_at)  
 VALUES (%s, %s, %s, %s, %s, toTimestamp(now()))  
 """,  
 (fgitem, image\_id, os.path.basename(image\_path), img\_bytes, emb.tolist()),  
 )  
 print(f"➕ Inserted new image for {fgitem} (image\_id={image\_id})")  
 return "inserted", image\_id, 1.0  
  
# 2) Text → image search across all items  
  
def search\_images\_by\_text(query: str, top\_k: int = 5):  
 q = embed\_text(query)  
 rows = session.execute("SELECT fgitem, image\_id, file\_name, embedding FROM fg\_item\_images")  
  
 scored = []  
 for r in rows:  
 sim = float(np.dot(q, np.array(r["embedding"], dtype=np.float32)))  
 scored.append({  
 "fgitem": r["fgitem"],  
 "image\_id": str(r["image\_id"]),  
 "file\_name": r["file\_name"],  
 "similarity": sim,  
 })  
  
 scored.sort(key=lambda x: x["similarity"], reverse=True)  
 return scored[:top\_k]  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Demo data (run once)  
 upsert\_item("FG001", "B001", 100, "SiteA")  
 upsert\_item("FG002", "B010", 40, "SiteB")  
  
 # Ingest images (place files in this folder before running)  
 upsert\_image\_for\_item("FG001", "sample1.png")  
 upsert\_image\_for\_item("FG001", "sample1\_variant.png") # will likely be FOUND if similar  
 upsert\_image\_for\_item("FG002", "sample2.png")  
  
 # Try text search  
 hits = search\_images\_by\_text("front label of FG001")  
 for h in hits:  
 print(h)

**Notes** - CLIP embeddings are **cross-modal**: you can search images by **text**. - For large datasets, prefer Astra’s **ANN vector search** (if enabled in your DB) instead of scanning in Python.

## 4) Node.js (Express) backend API

Navigate to backend/.

### 4.1 Initialize & install

npm init -y  
npm install express cors dotenv cassandra-driver

### 4.2 .env

ASTRA\_BUNDLE=../secure-connect-yourdb.zip  
ASTRA\_TOKEN=AstraCS:YOUR\_LONG\_TOKEN  
ASTRA\_KEYSPACE=your\_keyspace  
PORT=5000

### 4.3 server.js

import express from "express";  
import cors from "cors";  
import dotenv from "dotenv";  
import pkg from "cassandra-driver";  
const { Client, auth } = pkg;  
  
dotenv.config();  
  
const app = express();  
app.use(cors());  
app.use(express.json({ limit: "10mb" }));  
  
// Astra connection (CQL)  
const client = new Client({  
 cloud: { secureConnectBundle: process.env.ASTRA\_BUNDLE },  
 authProvider: new auth.PlainTextAuthProvider("token", process.env.ASTRA\_TOKEN),  
 keyspace: process.env.ASTRA\_KEYSPACE,  
});  
await client.connect();  
  
// 4.3.1 Search items by fgitem  
app.get("/api/items", async (req, res) => {  
 const { fgitem } = req.query;  
 if (!fgitem) return res.status(400).json({ error: "fgitem is required" });  
 try {  
 const result = await client.execute(  
 "SELECT fgitem, batch\_number, quantity, site, modified\_date FROM fg\_items WHERE fgitem = ?",  
 [fgitem],  
 { prepare: true }  
 );  
 res.json(result.rows);  
 } catch (e) {  
 console.error(e);  
 res.status(500).json({ error: "DB error" });  
 }  
});  
  
// 4.3.2 Get images for an item (returns base64 thumbnails)  
app.get("/api/images", async (req, res) => {  
 const { fgitem } = req.query;  
 if (!fgitem) return res.status(400).json({ error: "fgitem is required" });  
 try {  
 const result = await client.execute(  
 "SELECT image\_id, file\_name, image\_bytes, uploaded\_at FROM fg\_item\_images WHERE fgitem=? LIMIT 10",  
 [fgitem],  
 { prepare: true }  
 );  
 const data = result.rows.map(r => ({  
 image\_id: r.image\_id.toString(),  
 file\_name: r.file\_name,  
 uploaded\_at: r.uploaded\_at,  
 base64: Buffer.from(r.image\_bytes).toString("base64"),  
 mime: "image/png"  
 }));  
 res.json(data);  
 } catch (e) {  
 console.error(e);  
 res.status(500).json({ error: "DB error" });  
 }  
});  
  
// 4.3.3 Upsert image for an item (ingest & embed handled by Python OR future service)  
// (Optional) You can wire this to call a Python microservice for embedding.  
  
const PORT = process.env.PORT || 5000;  
app.listen(PORT, () => console.log(`API running http://localhost:${PORT}`));

For production, consider storing images in object storage (S3/GCS) and keep only URLs + embeddings in Astra.

## 5) React front-end (Inventory Search + images)

Navigate to frontend/ and create the app:

npx create-react-app inventory-search  
cd inventory-search  
npm install

### 5.1 src/App.js

import React, { useState } from "react";  
  
function App() {  
 const [fgitem, setFgitem] = useState("");  
 const [items, setItems] = useState([]);  
 const [images, setImages] = useState([]);  
 const [loading, setLoading] = useState(false);  
  
 const api = (path) => `http://localhost:5000${path}`; // Express base URL  
  
 const search = async () => {  
 if (!fgitem) return;  
 setLoading(true);  
 try {  
 const r1 = await fetch(api(`/api/items?fgitem=${encodeURIComponent(fgitem)}`));  
 const itemsJson = await r1.json();  
 setItems(itemsJson);  
  
 const r2 = await fetch(api(`/api/images?fgitem=${encodeURIComponent(fgitem)}`));  
 const imgs = await r2.json();  
 setImages(imgs);  
 } catch (e) {  
 console.error(e);  
 setItems([]);  
 setImages([]);  
 } finally {  
 setLoading(false);  
 }  
 };  
  
 return (  
 <div className="p-6" style={{ fontFamily: "Inter, system-ui, Arial" }}>  
 <h1>Inventory Search</h1>  
 <div style={{ display: "flex", gap: 8, marginBottom: 16 }}>  
 <input  
 value={fgitem}  
 onChange={(e) => setFgitem(e.target.value)}  
 placeholder="Enter FG Item"  
 style={{ padding: 8, flex: 1 }}  
 />  
 <button onClick={search} style={{ padding: "8px 16px" }}>Search</button>  
 </div>  
  
 {loading && <p>Loading...</p>}  
  
 {/\* Items Table \*/}  
 {items.length > 0 && (  
 <table border="1" cellPadding="8" style={{ width: "100%", marginBottom: 24 }}>  
 <thead>  
 <tr>  
 <th>FG Item</th>  
 <th>Batch</th>  
 <th>Qty</th>  
 <th>Site</th>  
 <th>Modified</th>  
 </tr>  
 </thead>  
 <tbody>  
 {items.map((row, idx) => (  
 <tr key={idx}>  
 <td>{row.fgitem}</td>  
 <td>{row.batch\_number}</td>  
 <td>{row.quantity}</td>  
 <td>{row.site}</td>  
 <td>{row.modified\_date}</td>  
 </tr>  
 ))}  
 </tbody>  
 </table>  
 )}  
  
 {/\* Images Gallery \*/}  
 {images.length > 0 && (  
 <div>  
 <h3>Images for {fgitem}</h3>  
 <div style={{ display: "grid", gridTemplateColumns: "repeat(auto-fill, 180px)", gap: 12 }}>  
 {images.map((img) => (  
 <div key={img.image\_id} style={{ border: "1px solid #ddd", padding: 8 }}>  
 <img  
 src={`data:${img.mime};base64,${img.base64}`}  
 alt={img.file\_name}  
 style={{ width: "100%", height: 120, objectFit: "contain" }}  
 />  
 <div style={{ fontSize: 12, marginTop: 4 }}>{img.file\_name}</div>  
 </div>  
 ))}  
 </div>  
 </div>  
 )}  
  
 {!loading && items.length === 0 && images.length === 0 && (  
 <p>No results yet. Try searching an FG Item.</p>  
 )}  
 </div>  
 );  
}  
  
export default App;

Run the React app:

npm start

It opens at **http://localhost:3000**.

## 6) Run order (end-to-end)

1. **Tables**: Create CQL tables in Astra (Section 2).
2. **Python**: Put sample1.png, sample1\_variant.png, sample2.png in python\_pipeline/ and run:

* python ingest\_and\_search.py
  + Confirms FOUND/INSERTED for images.

1. **Backend**: In backend/ start API

* node server.js

1. **Frontend**: In frontend/inventory-search/ run

* npm start

1. Search by **FG Item** (e.g., FG001). You’ll see item rows and any stored images.

## 7) (Optional) ANN vector search in Astra

If your Astra DB supports CQL ANN:

-- Example (adjust to your version)  
SELECT fgitem, image\_id, file\_name  
FROM fg\_item\_images  
ORDER BY embedding ANN OF [/\* 512 floats from query \*/]  
LIMIT 5;

From Python, you’d pass the query vector via a prepared statement.

## 8) Next steps & tips

* Store images in S3/GCS; keep only URL + embedding in DB.
* Add an **upload** API to let users add images from the React app.
* Add **audit/versioning** per item with status + approver.
* Add **PDF table** with text embeddings (1536 dims) for document search.

**You now have a complete, step-by-step blueprint** to stand up the DIG Assistant with Python image search, an Express API, and a React UI. Copy the code blocks into the indicated files and follow the run order above.