Product Expert System — Deployment, Ingestion & Testing Guide

---

Table of Contents

---

1. Prerequisites & Software Installation

1.1 Hardware Requirements

Component	Minimum	Recommended
CPU	4 cores	8+ cores
RAM	8 GB	16+ GB (Ollama LLM needs ~6 GB alone)
Disk	20 GB free	50+ GB (for LLM models + document storage)
GPU	Not required	NVIDIA GPU w/ 8+ GB VRAM (dramatically speeds up Ollama)

1.2 Required Software

Docker & Docker Compose (Required — Free)

Docker runs the entire stack (API, PostgreSQL, Redis, Nginx) in containers.

macOS:

```bash
# Install Docker Desktop (includes Docker Compose)
brew install --cask docker

# Start Docker Desktop from Applications, then verify:
docker --version          # Expect: Docker version 27.x+
docker compose version    # Expect: Docker Compose version v2.x+
```

Ubuntu/Debian Linux:

```bash
# Official Docker install
curl -fsSL https://get.docker.com | sudo sh

# Add your user to the docker group (avoid sudo for every command)
sudo usermod -aG docker $USER
newgrp docker

# Verify
docker --version
docker compose version
```

Windows:

```powershell
# Install Docker Desktop from https://www.docker.com/products/docker-desktop
# Enable WSL 2 backend during installation

# After install, open PowerShell and verify:
docker --version
docker compose version
```

> **License:** Docker Desktop is free for personal use and small businesses (<250 employees, <$10M revenue). Larger organizations need a Docker Business subscription ($24/user/month). Docker Engine on Linux is fully free and open source.

Git (Required — Free)

```bash
# macOS
brew install git

# Ubuntu/Debian
sudo apt-get install git

# Windows: download from https://git-scm.com/download/win

# Verify
git --version
```

GNU Make (Required — Free)

Used for operational shortcuts via the Makefile.

```bash
# macOS (included with Xcode CLI tools)
xcode-select --install

# Ubuntu/Debian
sudo apt-get install make

# Windows: install via chocolatey
choco install make
# Or use Git Bash which includes make
```

---

curl + jq (Required for testing — Free)

```bash
```

```
# macOS
brew install curl jq

# Ubuntu/Debian
sudo apt-get install curl jq

# Windows
choco install curl jq
```

Python 3.12+ (Optional — for local development without Docker)

Only needed if you want to run the API outside Docker.

```bash
# macOS
brew install python@3.12

# Ubuntu/Debian
sudo apt-get install python3.12 python3.12-venv python3.12-dev

# Create a virtual environment
python3.12 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

Node.js 20+ (Optional — for frontend development)

Only needed if building the React frontend as a standalone app (not the Claude artifact prototype).

```bash
```

```
# macOS
brew install node@20

# Ubuntu/Debian (via NodeSource)
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt-get install -y nodejs

# Verify
node --version   # v20.x+
npm --version    # 10.x+
```

1.3 Embedding & LLM Provider (Choose One)

The system needs two AI services: an **embedding model** (converts text to vectors for search) and an **LLM** (generates natural language answers for the Q&A feature). You have three options:

Option A: Ollama (Recommended for development — Free, runs locally)

Ollama runs open-source models on your own machine. No API keys, no costs, full privacy.

```bash
```

```
# ── Install Ollama ──────────────────────────────────────────

# macOS
brew install ollama

# Linux
curl -fsSL https://ollama.com/install.sh | sh

# Windows: download from https://ollama.com/download/windows

# ── Start the Ollama service ────────────────────────────────
ollama serve
# (runs on http://localhost:11434 — leave this terminal open)

# ── Pull required models (in a new terminal) ────────────────

# Embedding model (~275 MB download)
ollama pull nomic-embed-text

# LLM for RAG Q&A answers (~4.7 GB download)
ollama pull llama3.1:8b

# ── Verify models are available ─────────────────────────────
ollama list
# Should show:
#   nomic-embed-text    274 MB
#   llama3.1:8b         4.7 GB

# ── Quick test ──────────────────────────────────────────────
curl http://localhost:11434/api/embeddings -d '{
  "model": "nomic-embed-text",
  "prompt": "laboratory refrigerator"
}'
# Should return a JSON object with "embedding": [0.123, -0.456, ...]
```

> **License:** Ollama is MIT licensed (free). Models have their own licenses — nomic-embed-text is Apache 2.0 (free commercial use), Llama 3.1 is Meta's community license (free for <700M monthly active users).

.env settings for Ollama:

```env
```

```
EMBEDDING_PROVIDER=ollama
EMBEDDING_API_URL=http://host.docker.internal:11434/api/embeddings
EMBEDDING_MODEL=nomic-embed-text
LLM_PROVIDER=ollama
LLM_API_URL=http://host.docker.internal:11434/api/generate
LLM_MODEL=llama3.1:8b
```

> **Note:** `host.docker.internal` lets Docker containers reach Ollama running on your host machine. On Linux, you may need to use your machine's actual IP address or add `--add-host=host.docker.internal:host-gateway` to Docker run.

Option B: OpenAI API (Best quality, paid)

```bash
# Sign up at https://platform.openai.com
# Create an API key at https://platform.openai.com/api-keys
```

.env settings for OpenAI:

```env
EMBEDDING_PROVIDER=openai
OPENAI_API_KEY=sk-proj-...your-key...
EMBEDDING_MODEL=text-embedding-3-small

LLM_PROVIDER=openai
OPENAI_LLM_API_KEY=sk-proj-...your-key...
LLM_MODEL=gpt-4o-mini
```

> **Cost estimate:** text-embedding-3-small costs ~$0.02/1M tokens, gpt-4o-mini ~$0.15/1M input tokens. For this system's usage, expect $5–20/month depending on ingestion volume and Q&A usage.

> **License:** Proprietary API. Pay-per-use. No local installation needed.

Option C: Anthropic Claude API (Alternative paid option)

.env settings for Anthropic:

```env
```

```
LLM_PROVIDER=anthropic
ANTHROPIC_API_KEY=sk-ant-...your-key...
LLM_MODEL=claude-sonnet-4-20250514

# Still need an embedding provider (Anthropic doesn't offer embeddings)
EMBEDDING_PROVIDER=ollama
EMBEDDING_API_URL=http://host.docker.internal:11434/api/embeddings
EMBEDDING_MODEL=nomic-embed-text
```

> **License:** Proprietary API. Pay-per-use.

1.4 Complete Software Summary

Software	Purpose	License	Cost	Required?
Docker Desktop	Container runtime	Free / Paid for large orgs	Free–$24/user/mo	**Yes**
Docker Engine (Linux)	Container runtime	Apache 2.0	Free	**Yes** (Linux alternative)
Git	Version control	GPLv2	Free	**Yes**
GNU Make	Build automation	GPLv3	Free	**Yes**
curl + jq	API testing	MIT / MIT	Free	**Yes**
Ollama	Local AI models	MIT	Free	**Option A**
nomic-embed-text	Embedding model	Apache 2.0	Free	**Option A**
Llama 3.1 8B	LLM for Q&A	Meta Community	Free	**Option A**
OpenAI API	Cloud AI models	Proprietary	Pay-per-use	**Option B**
Anthropic API	Cloud LLM	Proprietary	Pay-per-use	**Option C**
Python 3.12	Local development	PSF License	Free	Optional
Node.js 20	Frontend dev	MIT	Free	Optional
PostgreSQL 16	Database	PostgreSQL License	Free	Via Docker

Software	Purpose	License	Cost	Required?
pgvector	Vector search extension	PostgreSQL License	Free	Via Docker
Redis 7	Cache & job queue	BSD 3-Clause	Free	Via Docker
Nginx	Reverse proxy	BSD 2-Clause	Free	Via Docker
FastAPI	Web framework	MIT	Free	Via Docker

Bottom line: The entire stack can run with zero paid licenses using Docker (Linux) + Ollama.

---

2. Project Directory Structure

After building all modules, your project should look like this:

```
product-expert/
├── .env                        # Environment variables (from .env.example)
├── .env.example                # Template
├── Dockerfile                  # Python API container
├── docker-compose.yml          # Full stack orchestration
├── Makefile                    # Operational shortcuts
├── requirements.txt            # Python dependencies
├── nginx.conf                  # Reverse proxy config
│
├── db-schema.sql               # 001 — PostgreSQL schema (15 tables)
├── models.py                   # 002 — Pydantic models & parsers
├── extraction_pipeline.py      # 003 — Document text extraction
├── ingestion_orchestrator.py   # 004 — Ingestion pipeline
├── recommendation_engine.py    # 005 — Recommendation scoring
├── rag_retrieval.py            # 006 — RAG retrieval & Q&A
├── api_layer.py                # 007 — FastAPI endpoints
├── asyncpg_repository.py       # 008 — Production DB repository
├── config.py                   # 009 — Pydantic Settings
├── seed.sql                    # 010 — Reference data seeding
│
├── alembic.ini                 # Migration configuration
├── alembic/
│   ├── env.py                  # Async migration runner
│   └── versions/
│       └── 001_initial_schema.py # Initial migration
│
├── data/                       # Document storage
│   ├── uploads/                # Ingested files
│   └── samples/                # Sample data sheets for testing
│       ├── ABS_Premier_26S_PDS.pdf
│       ├── ABS_Premier_47G_PDS.pdf
│       ├── LABRepCo_Futura_Silver_Specs.pdf
│       ├── Corepoint_NSF_Vaccine.txt
│       └── CBS_CryoSafe_2105.md
│
├── tests/                      # Test suite
│   ├── conftest.py
│   ├── test_extraction.py
│   ├── test_ingestion.py
│   ├── test_recommendation.py
│   ├── test_rag.py
│   └── test_api.py
│
```

```
└── frontend/                # React frontend (optional standalone)
    ├── dist/                # Built static files served by Nginx
    ├── src/
    └── package.json
```

3. Environment Configuration

```bash
# Copy the template
cp .env.example .env

# Edit with your preferred settings
nano .env   # or vim, code, etc.
```

Key settings to review:

```env
# Change the default password for production!
POSTGRES_PASSWORD=your-secure-password-here

# Match your AI provider choice (Section 1.3):
EMBEDDING_PROVIDER=ollama
LLM_PROVIDER=ollama

# For production, generate real API keys:
API_KEYS=your-admin-key-here:admin,your-sales-key:sales_engineer
```

4. Deployment — Step by Step

Step 1: Clone and enter the project

```bash
git clone https://github.com/your-org/product-expert.git
cd product-expert
```

Step 2: Configure environment

```bash
cp .env.example .env
# Edit .env — at minimum set POSTGRES_PASSWORD and your AI provider
```

Step 3: Start the AI provider (if using Ollama)

```bash
# In a separate terminal:
ollama serve

# Pull models (first time only):
ollama pull nomic-embed-text
ollama pull llama3.1:8b
```

Step 4: Build Docker images

```bash
make build
# This runs: docker compose build
# Takes 2-5 minutes on first run (downloads base images, installs Python deps)
```

Step 5: Start all services

```bash
make up
# This runs: docker compose up -d
# Starts: postgres → redis → api → nginx
```

Expected output:

```
✓ Network product-expert_backend     Created
✓ Container product-expert-db        Healthy
✓ Container product-expert-redis     Started
✓ Container product-expert-api       Healthy
✓ Container product-expert-proxy     Started

╔══════════════════════════════════════════╗
║  Product Expert System                   ║
║  Frontend:  http://localhost             ║
║  API:       http://localhost:8000        ║
║  API (via nginx): http://localhost/api/  ║
╚══════════════════════════════════════════╝
```

Step 6: Verify everything is running

```bash
# Check container status
make status
# All 4 containers should show "Up" and "healthy"

# Health check
make health
# Expected response:
# {
#   "status": "healthy",
#   "postgres": "connected",
#   "redis": "connected",
#   "embedding_service": "available"
# }
```

Step 7: Verify the database was seeded

```bash
```

```
make db-shell
# Opens psql inside the postgres container

# Run these checks:
SELECT COUNT(*) FROM brands;           -- Should be 5
SELECT COUNT(*) FROM product_families;  -- Should be 17
SELECT COUNT(*) FROM spec_registry;     -- Should be 30
SELECT COUNT(*) FROM model_patterns;    -- Should be 16
SELECT COUNT(*) FROM users;             -- Should be 4

# Exit psql:
\q
```

Step 8: Quick API smoke test

```bash
# List available use cases
curl -s http://localhost:8000/use-cases \
  -H "X-API-Key: dev-key-001" | jq .

# Search products (will be empty until ingestion)
curl -s "http://localhost:8000/products?limit=10" \
  -H "X-API-Key: dev-key-001" | jq .

# System stats
make stats
```

The system is now running and ready for document ingestion.

---

5. Document Ingestion

5.1 Preparing Documents

Place your product documents in the `data/samples/` directory. Supported formats:

Format	Extension	Best For
PDF	`.pdf`	Product Data Sheets, Cut Sheets, Performance Data
Plain Text	`.txt`	Extracted spec tables, feature lists
Markdown	`.md`	Structured product descriptions

Naming convention (recommended):

```
{Brand}_{ProductLine}_{Model}_{DocType}.{ext}

Examples:
  ABS_Premier_ABT-HC-26S_PDS.pdf
  LABRepCo_Futura_LRP-HC-RFC-2304G_CutSheet.pdf
  Corepoint_NSF_NSBR161WSW_Performance.pdf
  CBS_CryoSafe_2105-PA_Description.md
```

5.2 Ingesting a Single File

```bash
# Using curl directly
curl -X POST http://localhost:8000/ingest \
  -H "X-API-Key: dev-key-001" \
  -F "files=@data/samples/ABS_Premier_26S_PDS.pdf" | jq .

# Using the Makefile shortcut
make ingest-file FILE=data/samples/ABS_Premier_26S_PDS.pdf
```

Expected response:

```json
{
  "job_id": "a1b2c3d4-...",
  "status": "processing",
  "files_accepted": 1,
  "files_rejected": 0,
  "message": "1 file(s) queued for processing"
}
```

5.3 Bulk Ingestion (all files in data/ directory)

```bash
make ingest-sample
# Iterates over all .pdf, .txt, .md files in data/ and ingests each one
```

5.4 What Happens During Ingestion

The pipeline (modules 003–004) performs these steps automatically:

```
Upload received
  → SHA-256 hash check (skip duplicates)
  → Text extraction (PyMuPDF for PDFs)
  → Document type classification (PDS / Cut Sheet / Feature List / Performance / Cryo)
  → Brand detection (regex patterns for ABS, LABRepCo, Corepoint, °celsius, CBS)
  → Model number extraction (16 regex patterns)
  → Field mapping (40+ synonym → canonical name mappings)
  → Compound field parsing (door config, shelf config, temp range, electrical, etc.)
  → Product find-or-create in database
  → Spec conflict detection (5% numeric threshold for normal specs, 20% for non-critical)
  → Revision-aware update (newer revision dates win)
  → RAG chunking (section-based splitting, ~500 tokens per chunk)
  → Embedding generation (via Ollama/OpenAI)
  → Vector storage in pgvector
  → Document ↔ Product linkage
  → Spec registry auto-discovery (new fields flagged for review)
```

5.5 Monitoring Ingestion

```bash
# Check system stats (shows product/document/chunk counts)
make stats

# Check for any spec conflicts that need resolution
make conflicts
```

5.6 Resolving Spec Conflicts

When a new document disagrees with existing data (e.g., capacity listed as 47 cu.ft in one doc vs 49 cu.ft in another), a conflict is created.

```bash
# List pending conflicts
curl -s http://localhost:8000/conflicts \
  -H "X-API-Key: dev-key-001" | jq .

# Resolve a conflict — keep existing value
curl -X PUT http://localhost:8000/conflicts/{conflict_id} \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"resolution": "keep_existing"}'

# Accept the new value from the document
curl -X PUT http://localhost:8000/conflicts/{conflict_id} \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"resolution": "accept_new"}'

# Manual override with a corrected value
curl -X PUT http://localhost:8000/conflicts/{conflict_id} \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"resolution": "manual_override", "override_value": "48"}'

# Dismiss (not a real conflict)
curl -X PUT http://localhost:8000/conflicts/{conflict_id} \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"resolution": "dismissed"}'
```

---

6. Testing the System

6.1 API Endpoint Tests

After ingesting some documents, run through each endpoint:

Health & Stats

```bash
```

```
# Health check (no auth required)
curl -s http://localhost:8000/health | jq .

# System statistics
curl -s http://localhost:8000/stats \
  -H "X-API-Key: dev-key-001" | jq .
```

Product Search

```bash
# List all products
curl -s "http://localhost:8000/products?limit=20" \
  -H "X-API-Key: dev-key-001" | jq .

# Filter by brand
curl -s "http://localhost:8000/products?brand=ABS&limit=10" \
  -H "X-API-Key: dev-key-001" | jq .

# Filter by capacity range and door type
curl -s "http://localhost:8000/products?capacity_min=10&capacity_max=30&door_type=glass" \
  -H "X-API-Key: dev-key-001" | jq .

# Filter by certification
curl -s "http://localhost:8000/products?certifications=NSF/ANSI+456" \
  -H "X-API-Key: dev-key-001" | jq .

# Full-text search
curl -s "http://localhost:8000/products?text=vaccine+pharmacy" \
  -H "X-API-Key: dev-key-001" | jq .

# Single product lookup
curl -s http://localhost:8000/products/ABT-HC-26S \
  -H "X-API-Key: dev-key-001" | jq .
```

Recommendations

```bash
```

```
# Use-case driven recommendation
curl -s -X POST http://localhost:8000/recommend \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{
    "use_case": "vaccine_storage",
    "constraints": {
      "capacity_min_cuft": 10,
      "capacity_max_cuft": 20,
      "door_type": "glass",
      "voltage": 115,
      "certifications_required": ["NSF/ANSI 456"]
    },
    "max_results": 5
  }' | jq .

# Free-text recommendation
curl -s -X POST http://localhost:8000/recommend \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{
    "use_case_text": "I need a quiet undercounter fridge for chromatography, around 5 cubic feet, must fit under 34 inches",
    "max_results": 3
  }' | jq .

# Lab general with dimension constraints
curl -s -X POST http://localhost:8000/recommend \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{
    "use_case": "laboratory_general",
    "constraints": {
      "capacity_min_cuft": 20,
      "max_width_in": 30,
      "max_depth_in": 35,
      "max_height_in": 80
    }
  }' | jq .
```

Product Comparison

```
bash
```

```
# Compare 2-4 products side by side (use actual product IDs from your data)
curl -s -X POST http://localhost:8000/compare \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{
    "product_ids": ["<product-uuid-1>", "<product-uuid-2>"],
    "highlight_differences": true
  }' | jq .
```

RAG Q&A (requires LLM to be running)

```bash
# Spec lookup question
curl -s -X POST http://localhost:8000/ask \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"question": "What is the storage capacity of the ABT-HC-26S?"}' | jq .

# Comparison question
curl -s -X POST http://localhost:8000/ask \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"question": "Compare ABS and LABRepCo 23 cubic foot refrigerators"}' | jq .

# Compliance question
curl -s -X POST http://localhost:8000/ask \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"question": "Which refrigerators meet NSF/ANSI 456 for vaccine storage?"}' | jq .

# Recommendation question
curl -s -X POST http://localhost:8000/ask \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"question": "What is the best freezer for plasma storage under $5000?"}' | jq .
```

Cross-Brand Equivalents

```bash
```

```
curl -s http://localhost:8000/equivalents/ABT-HC-26S \
  -H "X-API-Key: dev-key-001" | jq .
```

6.2 Auth & RBAC Testing

```bash
# No API key — should get 401
curl -s http://localhost:8000/stats | jq .
# → {"detail": "Missing API key"}

# Invalid key — should get 403
curl -s http://localhost:8000/stats \
  -H "X-API-Key: fake-key" | jq .
# → {"detail": "Invalid API key"}

# Customer role trying to ingest — should get 403
curl -s -X POST http://localhost:8000/ingest \
  -H "X-API-Key: customer-key-001" \
  -F "files=@data/samples/test.pdf" | jq .
# → {"detail": "Insufficient permissions. Required: product_manager or admin"}

# Sales engineer role — can search and recommend, not ingest
curl -s http://localhost:8000/products \
  -H "X-API-Key: sales-key-001" | jq .
# → Works fine
```

6.3 Automated Test Suite

```bash
# Run all tests inside the Docker container
make test

# Run with coverage report
make test-cov

# Run locally (requires Python venv + PostgreSQL running)
make test-local
```

Test modules and what they cover:

Test File	Tests
`test_extraction.py`	PDF text extraction, field mapping, compound parsers (temp range, door config, shelf config, electrical, refrigerant, certifications, fractions)
`test_ingestion.py`	SHA-256 dedup, model number regex matching (all 16 patterns), product create/update, spec conflict detection thresholds, revision date parsing, RAG chunking
`test_recommendation.py`	Hard constraint filtering (voltage, door, certifications), soft scoring (capacity proximity, performance, efficiency), use-case profile matching, free-text resolution, equivalence detection
`test_rag.py`	Query parsing (model extraction, brand detection, spec synonym expansion, intent classification), vector search, keyword search, Reciprocal Rank Fusion, context building with token budget, grounding validation
`test_api.py`	All 12 endpoints, auth/RBAC, file upload validation, pagination, error handling

6.4 Load Testing (Optional)

```bash
# Install hey (HTTP load generator)
# macOS:
brew install hey

# Ubuntu:
sudo apt-get install hey

# Test product search endpoint (100 requests, 10 concurrent)
hey -n 100 -c 10 -H "X-API-Key: dev-key-001" \
  http://localhost:8000/products?limit=10

# Test recommendation endpoint
hey -n 50 -c 5 -m POST \
  -H "X-API-Key: dev-key-001" \
  -H "Content-Type: application/json" \
  -d '{"use_case": "laboratory_general", "max_results": 5}' \
  http://localhost:8000/recommend
```

7. Frontend Setup

Option A: Claude Artifact Prototype (already built)

The React frontend prototype runs directly in the Claude artifact viewer. To connect it to your running API, update the `API` constant at the top to point to your deployment:

```javascript
const API = "http://localhost:8000";  // For local Docker deployment
// or
const API = "https://your-domain.com/api";  // For production
```

Option B: Standalone React App

```bash
# Create a Vite React project
cd frontend
npm create vite@latest . -- --template react
npm install

# Install dependencies used by the prototype
npm install lucide-react recharts

# Copy the artifact code into src/App.jsx

# Development mode (hot reload)
npm run dev
# → http://localhost:5173

# Production build (output to dist/ for Nginx)
npm run build
# dist/ folder is mounted into Nginx container via docker-compose
```

---

8. Monitoring & Maintenance

Daily Operations

```bash
```

```bash
# Check system health
make health

# View API logs (real-time)
make logs

# View all service logs
make logs-all

# Check for pending conflicts
make conflicts

# System statistics
make stats
```

Database Maintenance

```bash
# Backup database
make db-dump
# Creates: backup_20260217_143022.sql

# Open psql for ad-hoc queries
make db-shell

# Example queries:
SELECT b.name, COUNT(p.id)
FROM brands b LEFT JOIN products p ON b.id = p.brand_id
GROUP BY b.name;

SELECT model_number, storage_capacity_cuft, certifications
FROM products
WHERE certifications @> ARRAY['NSF/ANSI 456']
ORDER BY storage_capacity_cuft;
```

Updating the Stack

```bash
```

```
# Pull latest code
git pull origin main

# Rebuild and restart
make build
make down
make up

# Run any new migrations
make migrate
```

Full Reset (Destructive)

bash

```bash
# Wipe everything and start fresh
make reset
# This destroys all data, rebuilds images, and starts clean
```

---

9. Troubleshooting

Container won't start

bash

```bash
# Check container logs
docker compose logs api
docker compose logs postgres

# Common issue: port already in use
# Fix: stop whatever is using port 5432, 6379, 8000, or 80
sudo lsof -i :8000
```

Database connection refused

bash

```
# Verify postgres is healthy
docker compose ps postgres
# Should show "healthy"

# If not, check postgres logs
docker compose logs postgres

# Common fix: remove stale volume and rebuild
docker compose down -v
docker compose up -d
```

Ollama connection fails from Docker

```bash
# On macOS/Windows Docker Desktop, host.docker.internal works automatically

# On Linux, use your machine's IP:
ip addr show docker0  # Usually 172.17.0.1
# Then set in .env:
EMBEDDING_API_URL=http://172.17.0.1:11434/api/embeddings
LLM_API_URL=http://172.17.0.1:11434/api/generate

# Or start Docker with:
docker compose up -d --add-host=host.docker.internal:host-gateway
```

Ingestion produces no products

```bash
```

```
# Check API logs for extraction errors
make logs | grep -i "error\|exception\|failed"

# Verify the document contains extractable text (not scanned images)
# For scanned PDFs, you'd need OCR (not included — add Tesseract if needed)

# Test extraction manually
make shell
python -c "
from extraction_pipeline import ExtractionPipeline
ep = ExtractionPipeline()
result = ep.extract_from_file('/app/data/uploads/your-file.pdf')
print(result)
"
```

Q&A returns empty answers

```bash
# Verify Ollama is running and the LLM model is loaded
curl http://localhost:11434/api/tags | jq .

# Verify chunks exist in the database
make db-shell
SELECT COUNT(*) FROM document_chunks;
SELECT COUNT(*) FROM document_chunks WHERE embedding IS NOT NULL;
# Both should be > 0

# If embeddings are null, Ollama may not have been available during ingestion
# Re-ingest after confirming Ollama is running:
make ingest-sample
```

---

10. License Summary

Component	License	Commercial Use	Notes
Product Expert System (our code)	Proprietary	Internal Horizon Scientific	All 10 modules
FastAPI	MIT	Yes	Free

Component	License	Commercial Use	Notes
PostgreSQL 16	PostgreSQL License	Yes	Free, very permissive
pgvector	PostgreSQL License	Yes	Free
Redis 7	BSD 3-Clause	Yes	Free (pre-v7.4, check SSPL for newer)
Nginx	BSD 2-Clause	Yes	Free (community edition)
Docker Engine	Apache 2.0	Yes	Free on Linux
Docker Desktop	Proprietary	Conditional	Free <250 employees
Ollama	MIT	Yes	Free
nomic-embed-text	Apache 2.0	Yes	Free
Llama 3.1	Meta Community License	Conditional	Free <700M MAU
OpenAI API	Proprietary	Yes	Pay-per-use
Anthropic API	Proprietary	Yes	Pay-per-use
PyMuPDF	AGPL-3.0	**Caution**	Free for open source; commercial license available from Artifex for proprietary use
Python	PSF License	Yes	Free
All pip packages	MIT/BSD/Apache	Yes	Free

> **Important:** PyMuPDF (used for PDF extraction) is AGPL-licensed, which requires releasing source code if distributed. For a purely internal deployment (not distributed), AGPL does not apply. If you plan to distribute the system, either purchase a commercial PyMuPDF license from Artifex Software, or switch to `pdfplumber` (MIT license) by uncommenting it in `requirements.txt`.

---

11. Windows-Specific Instructions

11.1 Can I Run Everything on Windows?

Yes. The entire stack runs on Windows 10/11 via Docker Desktop. Here's the compatibility matrix:

Component	Windows Support	Notes
Docker containers (API, Postgres, Redis, Nginx)	**Full**	Docker Desktop runs Linux containers on Windows via WSL 2
Ollama (embeddings + LLM)	**Full**	Native Windows installer available
`docker compose` commands	**Full**	Works in PowerShell, cmd, and Git Bash
`curl` for API testing	**Full**	Built into Windows 10+. Use `curl.exe` in PowerShell (to avoid the `Invoke-WebRequest` alias)
`Makefile`	**Partial**	Bash syntax (`for`, `sleep`, `$`) doesn't work natively. Use the PowerShell script `ops.ps1` instead
`jq` for JSON formatting	**Needs install**	`winget install jqlang.jq` or `choco install jq`. PowerShell alternative: `ConvertFrom-Json` (built into ops.ps1)

11.2 Windows Prerequisites Installation

Open **PowerShell as Administrator** and run:

```powershell
```

```
# ── 1. Docker Desktop (Required) ──────────────────────────────────
# Download and install from: https://www.docker.com/products/docker-desktop
# During install: Enable WSL 2 backend (recommended over Hyper-V)
# After install, restart your computer, then verify:
docker --version
docker compose version

# If WSL 2 is not installed, Docker Desktop will prompt you. Or manually:
wsl --install
# Restart, then set WSL 2 as default:
wsl --set-default-version 2

# ── 2. Git (Required) ─────────────────────────────────────────────
winget install Git.Git
# Or download from: https://git-scm.com/download/win
git --version

# ── 3. Ollama (Required for local AI — Option A) ──────────────────
# Download from: https://ollama.com/download/windows
# After install:
ollama serve                    # Start the service (leave this PowerShell window open)
# In a NEW PowerShell window:
ollama pull nomic-embed-text    # Embedding model (~275 MB)
ollama pull llama3.1:8b         # LLM for Q&A (~4.7 GB)
ollama list                     # Verify both models appear

# ── 4. Optional but helpful ───────────────────────────────────────
winget install jqlang.jq        # JSON formatter (used in curl examples)
```

11.3 Windows Deployment — Step by Step

Use **PowerShell** (not cmd.exe) for all commands.

```
powershell
```

```
# Step 1: Clone the project
git clone https://github.com/your-org/product-expert.git
cd product-expert

# Step 2: Configure environment
Copy-Item .env.example .env
# Edit .env with Notepad, VS Code, or:
notepad .env

# Step 3: Start Ollama (in a separate PowerShell window)
ollama serve

# Step 4: Build Docker images
docker compose build

# Step 5: Start all services
docker compose up -d

# Step 6: Verify
docker compose ps                          # All containers should be "Up"
curl.exe -s http://localhost:8000/health | jq .    # Or without jq, just curl.exe

# Step 7: Check database seeding
docker compose exec postgres psql -U expert -d product_expert -c "SELECT COUNT(*) FROM brands;"
```

11.4 Using ops.ps1 Instead of Makefile

The `ops.ps1` PowerShell script is a drop-in replacement for every Makefile target. Save it in your project root and use it from PowerShell:

```
powershell
```

```
# First time only — allow running local scripts
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

# Show all available commands
.\ops.ps1 help

# ── Docker lifecycle ──────────────────────────────────────────
.\ops.ps1 build                  # Build all containers
.\ops.ps1 up                     # Start everything
.\ops.ps1 down                   # Stop everything
.\ops.ps1 status                 # Container status
.\ops.ps1 logs                   # Follow API logs
.\ops.ps1 logs api               # Follow specific service
.\ops.ps1 logs-all               # Follow all logs

# ── Ingestion ─────────────────────────────────────────────────
.\ops.ps1 ingest-file .\data\samples\ABS_Premier_26S_PDS.pdf
.\ops.ps1 ingest-sample          # All files in data\

# ── Testing ───────────────────────────────────────────────────
.\ops.ps1 smoke-test             # Runs all 6 API checks in sequence
.\ops.ps1 test                   # Pytest inside container
.\ops.ps1 test-recommend         # Test recommendation endpoint
.\ops.ps1 test-ask "Which fridges meet NSF 456?"

# ── Monitoring ────────────────────────────────────────────────
.\ops.ps1 health                 # Health check
.\ops.ps1 stats                  # System statistics
.\ops.ps1 conflicts              # Pending spec conflicts
.\ops.ps1 products "vaccine"     # Search products

# ── Database ──────────────────────────────────────────────────
.\ops.ps1 db-shell               # Open psql
.\ops.ps1 db-dump                # Backup to SQL file
.\ops.ps1 db-reset               # Reset (with confirmation prompt)

# ── Dev access ────────────────────────────────────────────────
.\ops.ps1 shell                  # Bash inside API container
.\ops.ps1 python-shell           # Python REPL inside container
.\ops.ps1 redis-cli              # Redis CLI

# ── Cleanup ───────────────────────────────────────────────────
```

```
.\ops.ps1 clean                    # Remove containers
.\ops.ps1 reset                    # Full rebuild (with confirmation)
```

11.5 Windows curl — Important Note

PowerShell has a built-in alias `curl` that maps to `Invoke-WebRequest`, which behaves completely differently from real curl. Always use **`curl.exe`** (with the .exe suffix) to get the real curl:

```powershell
# WRONG — uses PowerShell's Invoke-WebRequest alias:
curl http://localhost:8000/health

# CORRECT — uses real curl:
curl.exe -s http://localhost:8000/health

# Or remove the alias for your session:
Remove-Item Alias:curl -ErrorAction SilentlyContinue
```

The `ops.ps1` script handles this automatically — it always calls `curl.exe`.

11.6 Windows Path Differences

When using `docker compose exec` or file paths:

```powershell
# Windows-style paths work for local files:
.\ops.ps1 ingest-file .\data\samples\ABS_Premier_26S_PDS.pdf

# But inside containers, always use Unix paths:
docker compose exec api cat /app/data/uploads/somefile.pdf

# Volume mounts in docker-compose.yml use forward slashes and work fine on Windows
```

11.7 Windows Troubleshooting

"Docker daemon is not running" → Start Docker Desktop from the Start Menu. Wait for the whale icon in the system tray to stop animating.

"WSL 2 installation is incomplete"

```powershell
```

```
wsl --install
# Restart your computer
wsl --set-default-version 2
```

Ollama can't be reached from Docker containers `host.docker.internal` works automatically on Docker Desktop for Windows. If it doesn't:

```powershell
# Check Ollama is running
curl.exe http://localhost:11434/api/tags

# If it works locally but not from Docker, check Windows Firewall
# Allow inbound connections on port 11434
```

Port conflicts (5432, 6379, 8000, 80)

```powershell
# Find what's using a port
netstat -ano | findstr :8000
# Kill the process
taskkill /PID <pid_number> /F

# Common conflict: local PostgreSQL on 5432
# Fix: change the host port mapping in docker-compose.yml:
#   ports:
#     - "5433:5432"   # Use 5433 on host instead
```

File upload fails with "permission denied" Docker Desktop for Windows sometimes has issues with volume mounts. Ensure you've shared the drive in Docker Desktop Settings → Resources → File Sharing.

Line ending issues (CRLF vs LF) If shell scripts or SQL files fail inside containers with strange errors:

```powershell
# Configure Git to check out with Unix line endings
git config core.autocrlf input

# Or fix a specific file
# In VS Code: click "CRLF" in the bottom status bar → select "LF"
```