## check\_db.py

"""  
Check database contents  
"""  
import sqlite3  
  
db\_path = "motherson\_graph.db"  
  
conn = sqlite3.connect(db\_path)  
cursor = conn.cursor()  
  
print("=" \* 70)  
print("DATABASE STATISTICS")  
print("=" \* 70)  
  
# Count tables  
tables = ['companies', 'divisions', 'facilities', 'events', 'sources', 'evidence', 'jobs']  
  
for table in tables:  
 cursor.execute(f"SELECT COUNT(\*) FROM {table}")  
 count = cursor.fetchone()[0]  
 print(f"{table:15} {count:5} rows")  
  
print("\n" + "=" \* 70)  
print("SAMPLE FACILITIES")  
print("=" \* 70)  
  
cursor.execute("""  
 SELECT f.name, d.name, f.city, f.state  
 FROM facilities f  
 JOIN divisions d ON f.division\_id = d.id  
 LIMIT 10  
""")  
  
for row in cursor.fetchall():  
 facility, division, city, state = row  
 print(f"{facility:30} | {division:20} | {city or 'N/A':15} | {state or 'N/A'}")  
  
print("\n" + "=" \* 70)  
print("SAMPLE EVENTS")  
print("=" \* 70)  
  
cursor.execute("""  
 SELECT f.name, e.event\_type, e.status, e.event\_date  
 FROM events e  
 JOIN facilities f ON e.facility\_id = f.id  
 LIMIT 10  
""")  
  
for row in cursor.fetchall():  
 facility, event\_type, status, event\_date = row  
 print(f"{facility:30} | {event\_type:15} | {status or 'N/A':20} | {event\_date or 'N/A'}")  
  
print("\n" + "=" \* 70)  
print("JOBS")  
print("=" \* 70)  
  
cursor.execute("SELECT title, location, is\_factory\_role FROM jobs LIMIT 10")  
jobs = cursor.fetchall()  
  
if jobs:  
 for row in jobs:  
 title, location, is\_factory = row  
 role\_type = "Factory" if is\_factory else "Non-factory"  
 print(f"{title:40} | {location or 'N/A':20} | {role\_type}")  
else:  
 print("No jobs found")  
  
conn.close()

## check\_db\_schema.py

"""  
Quick script to check if 'country' column exists in facilities table  
Run this after pipeline to verify schema  
"""  
import sqlite3  
  
db\_path = "motherson\_graph.db"  
  
conn = sqlite3.connect(db\_path)  
cursor = conn.cursor()  
  
# Check facilities table schema  
cursor.execute("PRAGMA table\_info(facilities)")  
columns = cursor.fetchall()  
  
print("=" \* 70)  
print("FACILITIES TABLE SCHEMA")  
print("=" \* 70)  
for col in columns:  
 print(f"{col[1]:20} {col[2]:15} {'NOT NULL' if col[3] else 'NULL'}")  
  
# Check if country column exists  
has\_country = any(col[1] == 'country' for col in columns)  
  
if not has\_country:  
 print("\n⚠️ WARNING: 'country' column NOT FOUND!")  
 print("Adding 'country' column with default 'India'...")  
   
 cursor.execute("ALTER TABLE facilities ADD COLUMN country TEXT DEFAULT 'India'")  
 cursor.execute("UPDATE facilities SET country = 'India' WHERE country IS NULL")  
 conn.commit()  
   
 print("✅ Added 'country' column")  
else:  
 print("\n✅ 'country' column exists")  
   
 # Check country distribution  
 cursor.execute("SELECT country, COUNT(\*) FROM facilities GROUP BY country")  
 countries = cursor.fetchall()  
   
 print("\nCountry Distribution:")  
 for country, count in countries:  
 print(f" {country or 'NULL':20} {count:5} facilities")  
  
conn.close()  
print("\n" + "=" \* 70)

## merge\_code.py

from docx import Document  
from pathlib import Path  
import os  
import subprocess  
  
# Create a new Word document  
doc = Document()  
  
# Define project root  
project\_root = Path(".") # run from project root folder  
  
# Define file extensions to include  
file\_extensions = [".py", ".txt", ".sh"]  
  
# Loop through files recursively  
for file\_path in project\_root.rglob("\*"):  
 # Skip .venv, \_\_pycache\_\_, and directories  
 if any(part in str(file\_path).split(os.sep) for part in [".venv", "\_\_pycache\_\_"]):  
 continue  
  
 if file\_path.suffix.lower() in file\_extensions:  
 try:  
 # Add file name as heading  
 doc.add\_heading(str(file\_path), level=2)  
 # Read file content  
 content = file\_path.read\_text(encoding="utf-8")  
 doc.add\_paragraph(content)  
 # Add a line break  
 doc.add\_paragraph("\n")  
 except Exception as e:  
 print(f"Skipping {file\_path} due to error: {e}")  
  
# Save Word document  
output\_file = "All\_Code.docx"  
doc.save(output\_file)  
print(f"✅ All code merged into {output\_file}")  
  
# Open Word file automatically (Windows)  
try:  
 subprocess.run(["start", output\_file], shell=True)  
except Exception as e:  
 print(f"Could not open file automatically: {e}")

## requirements.txt

# Core  
streamlit==1.31.0  
pandas==2.1.4  
numpy==1.24.3  
  
# NLP  
spacy==3.7.2  
python-dotenv==1.0.0  
  
# Vector DB  
chromadb==0.4.22  
sentence-transformers==2.3.1  
  
# LLM - UPDATED  
google-generativeai==0.4.0  
  
# Web Scraping  
requests==2.31.0  
beautifulsoup4==4.12.3  
trafilatura==1.6.3  
urllib3==2.1.0  
selenium==4.15.2  
webdriver-manager==4.0.1  
  
# PDF - ENHANCED  
pdfplumber==0.10.3  
PyPDF2==3.0.1  
camelot-py[cv]==0.11.0  
tabula-py==2.8.2  
  
# Utilities  
tqdm==4.66.1

## run.py

"""  
Enhanced Pipeline with Address Directory and Fixed Selenium  
"""  
import argparse  
import logging  
import json  
import os  
import shutil  
from datetime import datetime  
from pathlib import Path  
from typing import Dict, List  
  
from src.graph.database import Database  
from src.config import DB\_PATH, DEFAULT\_URLS, CACHE\_DIR, PDF\_DIR, CHROMA\_COLLECTION, FACTORY\_CAREER\_URL  
  
logging.basicConfig(  
 level=logging.INFO,  
 format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'  
)  
logger = logging.getLogger(\_\_name\_\_)  
  
# Import components  
from src.scrapers.base\_scraper import BaseScraper  
from src.scrapers.pdf\_extractor import PDFExtractor  
from src.scrapers.address\_scraper import AddressScraper # ✅ NEW  
from src.scrapers.selenium\_scraper import SeleniumScraper # ✅ FIXED  
from src.ner.entity\_extractor import EntityExtractor  
from src.graph.graph\_builder import GraphBuilder  
from src.rag.retriever import Retriever  
  
  
class EnhancedPipeline:  
 """Production pipeline with all fixes"""  
   
 def \_\_init\_\_(self):  
 self.db = Database(DB\_PATH)  
 self.scraper = BaseScraper()  
 self.pdf\_extractor = PDFExtractor()  
 self.address\_scraper = AddressScraper() # ✅ NEW  
 self.selenium\_scraper = SeleniumScraper() # ✅ FIXED  
 self.entity\_extractor = EntityExtractor()  
 self.graph\_builder = GraphBuilder(self.db)  
 self.retriever = Retriever()  
  
 def step1\_scraping(self, urls: list = None, use\_cache: bool = True) -> list:  
 """Step 1: Scrape all sources"""  
 logger.info("=" \* 70)  
 logger.info("STEP 1: SCRAPING (ENHANCED)")  
 logger.info("=" \* 70)  
  
 scraped\_data = []  
  
 if not use\_cache and os.path.exists(CACHE\_DIR):  
 shutil.rmtree(CACHE\_DIR)  
 os.makedirs(CACHE\_DIR, exist\_ok=True)  
 logger.info("Cache cleared.")  
  
 # ✅ STEP 1A: Scrape Address Directory (CRITICAL for Query 1)  
 logger.info("🏢 Scraping Address Directory (19 pages)...")  
 try:  
 address\_facilities = self.address\_scraper.scrape\_all(pages=19, use\_cache=use\_cache)  
   
 if address\_facilities:  
 # Convert to standard format  
 address\_doc = {  
 'url': 'https://www.motherson.com/contact/address-directory',  
 'title': 'Motherson Address Directory',  
 'text': json.dumps(address\_facilities, indent=2), # Store as JSON text  
 'fetched\_at': datetime.now().isoformat(),  
 'mime': 'application/json',  
 'publish\_dt': None,  
 'structured\_facilities': address\_facilities # ✅ Pass structured data  
 }  
 scraped\_data.append(address\_doc)  
 logger.info(f"✅ Address directory: {len(address\_facilities)} facilities")  
 else:  
 logger.warning("⚠️ No facilities from address directory")  
 except Exception as e:  
 logger.error(f"❌ Address directory scraping failed: {e}")  
  
 # ✅ STEP 1B: Scrape Career Page (CRITICAL for Query 3)  
 logger.info("👔 Scraping Factory Jobs...")  
 try:  
 jobs = self.selenium\_scraper.scrape\_jobs(url=FACTORY\_CAREER\_URL)  
   
 if jobs:  
 # Convert to standard format  
 jobs\_text = "\n\n".join([f"{j['title']} - {j['location']}" for j in jobs])  
 jobs\_doc = {  
 'url': FACTORY\_CAREER\_URL,  
 'title': 'Motherson Factory Jobs',  
 'text': jobs\_text,  
 'fetched\_at': datetime.now().isoformat(),  
 'mime': 'text/html',  
 'publish\_dt': None,  
 'structured\_jobs': jobs # ✅ Pass structured data  
 }  
 scraped\_data.append(jobs\_doc)  
 logger.info(f"✅ Career page: {len(jobs)} jobs")  
 else:  
 logger.warning("⚠️ No jobs from career page")  
 except Exception as e:  
 logger.error(f"❌ Career page scraping failed: {e}")  
  
 # ✅ STEP 1C: Scrape PDFs (CRITICAL for Query 2)  
 pdf\_files = list(PDF\_DIR.glob("\*.pdf"))  
 if pdf\_files:  
 logger.info(f"📄 Found {len(pdf\_files)} PDF files")  
 for pdf\_file in pdf\_files:  
 logger.info(f"Extracting: {pdf\_file.name}")  
 pdf\_data = self.pdf\_extractor.extract\_from\_file(str(pdf\_file))  
 if pdf\_data:  
 scraped\_data.append(pdf\_data)  
 else:  
 logger.warning("⚠️ No PDFs found in data/pdfs/")  
  
 # ✅ STEP 1D: Scrape Other URLs (optional)  
 other\_urls = [  
 "https://www.motherson.com/about-us",  
 "https://www.motherson.com/company/business-divisions",  
 ]  
   
 for url in other\_urls:  
 try:  
 data = self.scraper.scrape\_url(url, use\_cache=use\_cache)  
 if data:  
 scraped\_data.append(data)  
 except Exception as e:  
 logger.error(f"Failed {url}: {e}")  
  
 # Save cache  
 cache\_file = CACHE\_DIR / "scraped\_data.json"  
 with open(cache\_file, 'w', encoding='utf-8') as f:  
 json.dump(scraped\_data, f, indent=2, default=str)  
  
 logger.info(f"✅ Total documents scraped: {len(scraped\_data)}")  
 return scraped\_data  
  
 def step2\_extraction(self, scraped\_data: list) -> list:  
 """Step 2: Extract entities"""  
 logger.info("=" \* 70)  
 logger.info("STEP 2: ENTITY EXTRACTION")  
 logger.info("=" \* 70)  
  
 extracted\_data = []  
  
 for idx, doc in enumerate(scraped\_data, 1):  
 logger.info(f"Processing document {idx}/{len(scraped\_data)}: {doc.get('title', 'Untitled')}")  
  
 text = doc.get('text', '')  
 if not text or len(text) < 50:  
 logger.warning(" ⚠️ Document too short, skipping")  
 continue  
  
 try:  
 entities = self.entity\_extractor.extract\_entities(text)  
  
 extracted\_item = {  
 'source\_data': {  
 'url': doc.get('url', 'unknown'),  
 'title': doc.get('title', 'Untitled'),  
 'fetched\_at': doc.get('fetched\_at'),  
 'mime': doc.get('mime', 'text/html'),  
 'publish\_dt': doc.get('publish\_dt'),  
 'source\_type': 'pdf' if doc.get('mime') == 'application/pdf' else 'web'  
 },  
 'entities': entities  
 }  
  
 # ✅ Pass structured data  
 if 'structured\_facilities' in doc:  
 extracted\_item['source\_data']['structured\_facilities'] = doc['structured\_facilities']  
 logger.info(f" ✅ Found {len(doc['structured\_facilities'])} structured facilities")  
  
 if 'structured\_jobs' in doc:  
 extracted\_item['source\_data']['structured\_jobs'] = doc['structured\_jobs']  
 logger.info(f" ✅ Found {len(doc['structured\_jobs'])} structured jobs")  
  
 extracted\_data.append(extracted\_item)  
  
 total\_entities = sum(len(v) for v in entities.values())  
 logger.info(f" ✅ Extracted {total\_entities} entities")  
  
 except Exception as e:  
 logger.error(f" ✗ Error extracting from {doc.get('url')}: {e}")  
 import traceback  
 traceback.print\_exc()  
  
 cache\_file = CACHE\_DIR / "extracted\_data.json"  
 with open(cache\_file, 'w', encoding='utf-8') as f:  
 json.dump(extracted\_data, f, indent=2, default=str)  
  
 logger.info(f"✅ Extracted entities from {len(extracted\_data)} documents")  
 return extracted\_data  
  
 def step3\_graph\_building(self, extracted\_data: list):  
 """Step 3: Build knowledge graph"""  
 logger.info("=" \* 70)  
 logger.info("STEP 3: GRAPH BUILDING")  
 logger.info("=" \* 70)  
  
 try:  
 self.graph\_builder.build\_graph(extracted\_data)  
  
 # ✅ Insert jobs  
 logger.info("Extracting job postings...")  
 jobs\_data = []  
  
 for item in extracted\_data:  
 source\_data = item.get('source\_data', {})  
   
 # Get source\_id  
 source\_id = None  
 try:  
 result = self.graph\_builder.db.execute\_query(  
 "SELECT id FROM sources WHERE url = ?",  
 (source\_data.get('url', 'unknown'),)  
 )  
 if result:  
 source\_id = result[0][0]  
 except:  
 pass  
  
 # Use structured jobs if available  
 if 'structured\_jobs' in source\_data:  
 for job in source\_data['structured\_jobs']:  
 jobs\_data.append({  
 'title': job.get('title', ''),  
 'location': job.get('location', 'India'),  
 'division': None,  
 'is\_factory\_role': True, # Already filtered  
 'source\_id': source\_id,  
 'posted\_date': None,  
 'description': None  
 })  
  
 if jobs\_data:  
 self.graph\_builder.insert\_jobs(jobs\_data)  
 logger.info(f"✅ Inserted {len(jobs\_data)} job postings")  
  
 logger.info("✅ Graph building complete!")  
  
 except Exception as e:  
 logger.error(f"✗ Graph building failed: {e}")  
 import traceback  
 traceback.print\_exc()  
 raise  
  
 def step4\_vector\_indexing(self, scraped\_data: list):  
 """Step 4: Build vector index"""  
 logger.info("=" \* 70)  
 logger.info("STEP 4: VECTOR INDEXING")  
 logger.info("=" \* 70)  
  
 try:  
 self.retriever.index\_documents(scraped\_data)  
 count = self.retriever.collection.count()  
 logger.info(f"✅ Indexed {count} document chunks")  
 except Exception as e:  
 logger.error(f"✗ Vector indexing failed: {e}")  
 import traceback  
 traceback.print\_exc()  
 raise  
  
 def run\_full\_pipeline(self, urls: list = None, use\_cache: bool = True):  
 """Run complete pipeline"""  
 start\_time = datetime.now()  
  
 logger.info("=" \* 70)  
 logger.info("🚀 MOTHERSON INTELLIGENCE PIPELINE (PRODUCTION)")  
 logger.info("=" \* 70)  
 logger.info(f"Started at: {start\_time.strftime('%Y-%m-%d %H:%M:%S')}")  
  
 try:  
 scraped\_data = self.step1\_scraping(urls, use\_cache)  
 if not scraped\_data:  
 logger.error("❌ No data scraped. Exiting.")  
 return False  
  
 print()  
 extracted\_data = self.step2\_extraction(scraped\_data)  
 if not extracted\_data:  
 logger.error("❌ No entities extracted. Exiting.")  
 return False  
  
 print()  
 self.step3\_graph\_building(extracted\_data)  
  
 print()  
 self.step4\_vector\_indexing(scraped\_data)  
  
 elapsed = (datetime.now() - start\_time).total\_seconds()  
  
 logger.info("")  
 logger.info("=" \* 70)  
 logger.info("✅ PIPELINE COMPLETE!")  
 logger.info("=" \* 70)  
 logger.info(f"Total time: {elapsed:.2f} seconds ({elapsed/60:.1f} minutes)")  
 logger.info("")  
 logger.info("Next steps:")  
 logger.info(" 1. Run: streamlit run src/ui/app.py")  
 logger.info(" 2. Open browser at: http://localhost:8501")  
 logger.info(" 3. Try the 3 preset queries!")  
 logger.info("")  
  
 return True  
  
 except Exception as e:  
 logger.error(f"❌ Pipeline failed: {e}")  
 import traceback  
 traceback.print\_exc()  
 return False  
  
 def show\_statistics(self):  
 """Show database statistics"""  
 logger.info("=" \* 70)  
 logger.info("DATABASE STATISTICS")  
 logger.info("=" \* 70)  
  
 stats = self.db.get\_statistics()  
  
 print("\n📊 Graph Database:")  
 for key, value in stats.items():  
 print(f" {key}: {value}")  
  
 print("\n🔍 Vector Store:")  
 count = self.retriever.collection.count()  
 print(f" Document chunks: {count}")  
  
 print("\n💾 Cache:")  
 scraped\_file = CACHE\_DIR / "scraped\_data.json"  
 extracted\_file = CACHE\_DIR / "extracted\_data.json"  
  
 if scraped\_file.exists():  
 with open(scraped\_file, 'r') as f:  
 scraped\_count = len(json.load(f))  
 print(f" Scraped documents: {scraped\_count}")  
  
 if extracted\_file.exists():  
 with open(extracted\_file, 'r') as f:  
 extracted\_count = len(json.load(f))  
 print(f" Extracted documents: {extracted\_count}")  
  
 print("")  
  
  
def main():  
 parser = argparse.ArgumentParser(description="Motherson Intelligence Pipeline (Production)")  
 parser.add\_argument('--ingest', action='store\_true', help='Run full ingestion pipeline')  
 parser.add\_argument('--urls', nargs='+', help='URLs to scrape (optional)')  
 parser.add\_argument('--no-cache', action='store\_true', help='Disable cache, force fresh scraping')  
 parser.add\_argument('--stats', action='store\_true', help='Show database statistics')  
  
 args = parser.parse\_args()  
  
 pipeline = EnhancedPipeline()  
  
 if args.stats:  
 pipeline.show\_statistics()  
 elif args.ingest:  
 use\_cache = not args.no\_cache  
 success = pipeline.run\_full\_pipeline(urls=args.urls, use\_cache=use\_cache)  
  
 if success:  
 print("\n" + "=" \* 70)  
 print("🎉 SUCCESS! Ready to use the platform.")  
 print("=" \* 70)  
 else:  
 print("\n" + "=" \* 70)  
 print("❌ FAILED! Check logs above for errors.")  
 print("=" \* 70)  
 else:  
 parser.print\_help()  
 print("\n" + "=" \* 70)  
 print("Quick Start:")  
 print(" python run.py --ingest --no-cache # Run full pipeline with fresh data")  
 print(" python run.py --stats # Show statistics")  
 print(" streamlit run src/ui/app.py # Start web app")  
 print("=" \* 70)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 try:  
 from src.config import BASE\_DIR  
 if os.getcwd() != str(BASE\_DIR):  
 os.chdir(BASE\_DIR)  
 except Exception:  
 pass  
  
 main()

## setup.sh

#!/bin/bash  
  
echo "🚀 Setting up Motherson Intelligence Platform..."  
  
# Create virtual environment  
python3 -m venv venv  
source venv/bin/activate  
  
# Upgrade pip  
pip install --upgrade pip  
  
# Install requirements  
pip install -r requirements.txt  
  
# Download spaCy model  
python -m spacy download en\_core\_web\_sm  
  
# Create data directories  
mkdir -p data/cache data/vector\_store data/pdfs  
  
# Copy .env.example to .env  
if [ ! -f .env ]; then  
 cp .env.example .env  
 echo "⚠️ Please edit .env and add your GEMINI\_API\_KEY"  
fi  
  
echo "✅ Setup complete!"  
echo "Next steps:"  
echo "1. Edit .env and add your GEMINI\_API\_KEY"  
echo "2. Run: python run.py --ingest"  
echo "3. Run: streamlit run src/ui/app.py"

## src\config.py

"""Configuration settings - FIXED VERSION"""  
import os  
from pathlib import Path  
from dotenv import load\_dotenv  
  
load\_dotenv()  
  
# Base paths  
BASE\_DIR = Path(\_\_file\_\_).parent.parent.resolve()  
DATA\_DIR = BASE\_DIR / "data"  
CACHE\_DIR = DATA\_DIR / "cache"  
VECTOR\_STORE\_DIR = DATA\_DIR / "vector\_store"  
PDF\_DIR = DATA\_DIR / "pdfs"  
  
# Ensure directories exist  
for d in [DATA\_DIR, CACHE\_DIR, VECTOR\_STORE\_DIR, PDF\_DIR]:  
 d.mkdir(parents=True, exist\_ok=True)  
  
# Database  
DB\_PATH = str(BASE\_DIR / "motherson\_graph.db")  
  
# API Keys  
GEMINI\_API\_KEY = os.getenv("GEMINI\_API\_KEY", "")  
  
# Scraping settings  
SCRAPER\_RATE\_LIMIT = float(os.getenv("SCRAPER\_RATE\_LIMIT", 2.0))  
SCRAPER\_TIMEOUT = int(os.getenv("SCRAPER\_TIMEOUT", 30))  
SCRAPER\_MAX\_RETRIES = int(os.getenv("SCRAPER\_MAX\_RETRIES", 3))  
  
# ✅ ADDED: Factory career URL (CRITICAL FIX)  
FACTORY\_CAREER\_URL = "https://careers.motherson.com/en/jobs?area=EnvironmentHealthandSafety&area=LogisticsShippingScheduling&area=ManufacturingOperations&area=Purchasing&area=Quality"  
  
# Default URLs - UPDATED with address directory  
DEFAULT\_URLS = [  
 "https://motherson.com",  
 "https://www.motherson.com/about-us",  
 "https://www.motherson.com/our-businesses",  
 "https://www.motherson.com/company/business-divisions",  
 # ✅ Address directory pages (19 pages - CRITICAL for Query 1)  
 \*[f"https://www.motherson.com/contact/address-directory?country=India&page={i}" for i in range(1, 20)],  
 # ✅ Pre-filtered career URL (CRITICAL for Query 3)  
 FACTORY\_CAREER\_URL,  
]  
  
# NER / Vector settings  
SPACY\_MODEL = os.getenv("SPACY\_MODEL", "en\_core\_web\_sm")  
CHROMA\_COLLECTION = os.getenv("CHROMA\_COLLECTION", "motherson\_docs")  
CHUNK\_SIZE = int(os.getenv("CHUNK\_SIZE", 500))  
CHUNK\_OVERLAP = int(os.getenv("CHUNK\_OVERLAP", 50))  
  
# Query defaults  
EXPANSION\_MONTHS\_DEFAULT = int(os.getenv("EXPANSION\_MONTHS\_DEFAULT", 24))

## src\\_\_init\_\_.py

# ==================================================  
# File: src/\_\_init\_\_.py  
# ==================================================  
"""  
Motherson Intelligence System  
Main package initialization  
"""  
  
\_\_version\_\_ = "1.0.0"  
\_\_author\_\_ = "C4Scale Takehome"

## src\graph\database.py

"""  
Database - SQLite schema and operations  
Creates tables: companies, divisions, facilities, events, sources, evidence, jobs  
"""  
  
import sqlite3  
import logging  
from typing import List, Dict, Optional, Tuple  
from pathlib import Path  
  
from ..config import DB\_PATH  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class Database:  
 def \_\_init\_\_(self, path: str = DB\_PATH):  
 self.db\_path = path  
 self.conn = sqlite3.connect(path, check\_same\_thread=False)  
 self.conn.row\_factory = sqlite3.Row  
 self.cursor = self.conn.cursor()  
  
 # Ensure schema exists (idempotent)  
 self.\_create\_tables()  
   
 def \_create\_tables(self):  
 """Create minimal DB schema if missing (safe to call repeatedly)."""  
 schema = """  
 CREATE TABLE IF NOT EXISTS companies (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 name TEXT UNIQUE NOT NULL  
 );  
  
 CREATE TABLE IF NOT EXISTS divisions (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 company\_id INTEGER NOT NULL,  
 name TEXT NOT NULL,  
 FOREIGN KEY(company\_id) REFERENCES companies(id)  
 );  
  
 CREATE TABLE IF NOT EXISTS sources (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 url TEXT UNIQUE,  
 title TEXT,  
 fetched\_at TEXT,  
 mime\_type TEXT,  
 publish\_date TEXT,  
 source\_type TEXT  
 );  
  
 CREATE TABLE IF NOT EXISTS facilities (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 division\_id INTEGER,  
 name TEXT,  
 city TEXT,  
 state TEXT,  
 country TEXT,  
 normalized\_name TEXT,  
 FOREIGN KEY(division\_id) REFERENCES divisions(id)  
 );  
  
 CREATE TABLE IF NOT EXISTS events (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 facility\_id INTEGER,  
 event\_type TEXT,  
 event\_date TEXT,  
 status TEXT,  
 expansion\_type TEXT,  
 confidence REAL,  
 FOREIGN KEY(facility\_id) REFERENCES facilities(id)  
 );  
  
 CREATE TABLE IF NOT EXISTS evidence (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 source\_id INTEGER,  
 entity\_type TEXT,  
 entity\_id INTEGER,  
 text\_snippet TEXT,  
 char\_start INTEGER,  
 char\_end INTEGER,  
 confidence REAL,  
 FOREIGN KEY(source\_id) REFERENCES sources(id)  
 );  
  
 CREATE TABLE IF NOT EXISTS jobs (  
 id INTEGER PRIMARY KEY AUTOINCREMENT,  
 facility\_id INTEGER,  
 title TEXT,  
 location TEXT,  
 division\_id INTEGER,  
 is\_factory\_role INTEGER DEFAULT 0,  
 source\_id INTEGER,  
 posted\_date TEXT,  
 description TEXT,  
 FOREIGN KEY(facility\_id) REFERENCES facilities(id),  
 FOREIGN KEY(division\_id) REFERENCES divisions(id),  
 FOREIGN KEY(source\_id) REFERENCES sources(id)  
 );  
 """  
 self.conn.executescript(schema)  
 self.conn.commit()  
 logger.info(f"Database initialized at {self.db\_path}")  
   
 def get\_connection(self) -> sqlite3.Connection:  
 """Get database connection"""  
 return sqlite3.connect(self.db\_path)  
   
 def execute\_query(self, query: str, params: Tuple = ()) -> List[Tuple]:  
 """Execute SELECT query"""  
 conn = self.get\_connection()  
 cursor = conn.cursor()  
 cursor.execute(query, params)  
 results = cursor.fetchall()  
 conn.close()  
 return results  
   
 def execute\_insert(self, query: str, params: Tuple = ()) -> int:  
 """Execute INSERT and return last row id"""  
 conn = self.get\_connection()  
 cursor = conn.cursor()  
 cursor.execute(query, params)  
 last\_id = cursor.lastrowid  
 conn.commit()  
 conn.close()  
 return last\_id  
   
 def execute\_many(self, query: str, params\_list: List[Tuple]):  
 """Execute multiple INSERT/UPDATE statements"""  
 conn = self.get\_connection()  
 cursor = conn.cursor()  
 cursor.executemany(query, params\_list)  
 conn.commit()  
 conn.close()  
   
 def get\_statistics(self) -> Dict:  
 """Get database statistics"""  
 conn = self.get\_connection()  
 cursor = conn.cursor()  
   
 stats = {}  
 tables = ['companies', 'divisions', 'facilities', 'events', 'sources', 'evidence', 'jobs']  
   
 for table in tables:  
 cursor.execute(f"SELECT COUNT(\*) FROM {table}")  
 stats[f'total\_{table}'] = cursor.fetchone()[0]  
   
 conn.close()  
 return stats  
   
 def clear\_all\_data(self):  
 """Clear all data (for testing)"""  
 conn = self.get\_connection()  
 cursor = conn.cursor()  
   
 tables = ['evidence', 'jobs', 'events', 'facilities', 'divisions', 'companies', 'sources']  
 for table in tables:  
 cursor.execute(f"DELETE FROM {table}")  
   
 conn.commit()  
 conn.close()  
 logger.info("All data cleared")

## src\graph\entity\_resolver.py

"""  
Entity Resolver - IMPROVED VERSION  
Merge duplicate facilities via normalized name + location  
"""  
  
import re  
import logging  
from typing import List, Dict, Tuple, Optional  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class EntityResolver:  
 def \_\_init\_\_(self):  
 self.common\_suffixes = ['plant', 'facility', 'unit', 'manufacturing', 'factory', 'site']  
   
 def normalize\_name(self, name: str) -> str:  
 """  
 Normalize facility name for deduplication  
 Example: "Sanand Plant" -> "sanand"  
 """  
 if not name:  
 return ""  
   
 name = name.lower().strip()  
 name = re.sub(r'\s+', ' ', name) # Normalize whitespace  
 name = re.sub(r'[^\w\s]', '', name) # Remove punctuation  
   
 # Remove common suffixes  
 for suffix in self.common\_suffixes:  
 if name.endswith(' ' + suffix):  
 name = name[:-len(suffix)].strip()  
   
 return name  
   
 def normalize\_location(self, city: Optional[str], state: Optional[str]) -> str:  
 """  
 Normalize location for matching  
 Example: "Sanand, Gujarat" -> "sanand gujarat"  
 """  
 parts = []  
 if city:  
 parts.append(city.lower().strip())  
 if state:  
 parts.append(state.lower().strip())  
 return ' '.join(parts)  
   
 def calculate\_similarity(self, name1: str, name2: str) -> float:  
 """  
 Simple token-based similarity  
 Returns: 0.0 to 1.0  
 """  
 if not name1 or not name2:  
 return 0.0  
   
 tokens1 = set(name1.lower().split())  
 tokens2 = set(name2.lower().split())  
   
 if not tokens1 or not tokens2:  
 return 0.0  
   
 intersection = tokens1.intersection(tokens2)  
 union = tokens1.union(tokens2)  
   
 return len(intersection) / len(union) if union else 0.0  
   
 def should\_merge(self,  
 name1: str, loc1: str,  
 name2: str, loc2: str,  
 threshold: float = 0.8) -> bool: # Increased threshold to 0.8 for stricter merging  
 """  
 Decide if two facilities should be merged - STRICTER VERSION  
   
 Rules:  
 1. Exact normalized name match + same location -> merge  
 2. Very high name similarity (0.8+) + same state -> merge  
 3. Otherwise -> don't merge  
 """  
 norm\_name1 = self.normalize\_name(name1)  
 norm\_name2 = self.normalize\_name(name2)  
   
 # Rule 1: Exact name match  
 if norm\_name1 == norm\_name2 and norm\_name1: # Must not be empty  
 # Check location overlap  
 loc1\_lower = loc1.lower() if loc1 else ""  
 loc2\_lower = loc2.lower() if loc2 else ""  
   
 # If either location is empty, require additional check  
 if not loc1\_lower or not loc2\_lower:  
 # Only merge if names are longer than 5 chars  
 if len(norm\_name1) > 5:  
 return True  
 return False  
   
 # Check if locations overlap (any common tokens)  
 loc1\_tokens = set(loc1\_lower.split())  
 loc2\_tokens = set(loc2\_lower.split())  
   
 if loc1\_tokens.intersection(loc2\_tokens):  
 return True  
   
 # Rule 2: Very high name similarity + location match  
 name\_sim = self.calculate\_similarity(norm\_name1, norm\_name2)  
   
 if name\_sim >= threshold:  
 loc1\_lower = loc1.lower() if loc1 else ""  
 loc2\_lower = loc2.lower() if loc2 else ""  
   
 if loc1\_lower and loc2\_lower:  
 # Check for state-level match  
 states = ['gujarat', 'tamil nadu', 'maharashtra', 'haryana', 'karnataka',  
 'uttar pradesh', 'rajasthan', 'punjab', 'telangana']  
   
 for state in states:  
 if state in loc1\_lower and state in loc2\_lower:  
 return True  
   
 return False  
   
 def resolve\_facilities(self, facilities: List[Dict]) -> List[Dict]:  
 """  
 Resolve duplicate facilities - IMPROVED VERSION  
 Returns: Deduplicated list with merged entries  
 """  
 if not facilities:  
 return []  
   
 # Group by division first (only merge within same division)  
 division\_groups = {}  
 for fac in facilities:  
 div = fac.get('division', 'Unknown')  
 if div not in division\_groups:  
 division\_groups[div] = []  
 division\_groups[div].append(fac)  
   
 # Resolve within each division  
 resolved = []  
   
 for division, fac\_list in division\_groups.items():  
 merged\_indices = set()  
   
 for i, fac1 in enumerate(fac\_list):  
 if i in merged\_indices:  
 continue  
   
 # Find all facilities that should merge with fac1  
 merge\_group = [fac1]  
   
 for j, fac2 in enumerate(fac\_list):  
 if j <= i or j in merged\_indices:  
 continue  
   
 name1 = fac1.get('name', '')  
 name2 = fac2.get('name', '')  
   
 loc1 = self.normalize\_location(fac1.get('city'), fac1.get('state'))  
 loc2 = self.normalize\_location(fac2.get('city'), fac2.get('state'))  
   
 if self.should\_merge(name1, loc1, name2, loc2):  
 merge\_group.append(fac2)  
 merged\_indices.add(j)  
   
 # Create merged facility (use best data from group)  
 merged\_fac = self.\_merge\_facility\_group(merge\_group)  
 resolved.append(merged\_fac)  
   
 logger.info(f"Resolved {len(facilities)} facilities to {len(resolved)} unique facilities")  
 return resolved  
   
 def \_merge\_facility\_group(self, facilities: List[Dict]) -> Dict:  
 """  
 Merge a group of duplicate facilities  
 Strategy: Keep most complete data, prefer non-null values  
 """  
 merged = {}  
   
 # Take first facility as base  
 merged.update(facilities[0])  
   
 # Override with better data from other facilities  
 for fac in facilities[1:]:  
 for key, value in fac.items():  
 # If current value is None/empty and new value exists, use it  
 if not merged.get(key) and value:  
 merged[key] = value  
   
 # For dates, keep the earliest  
 if key in ['event\_date', 'announcement\_date'] and value:  
 if not merged.get(key) or value < merged[key]:  
 merged[key] = value  
   
 # For confidence, keep the highest  
 if key == 'confidence' and value:  
 if not merged.get(key) or value > merged[key]:  
 merged[key] = value  
   
 # Mark as merged  
 merged['was\_merged'] = True  
 merged['merge\_count'] = len(facilities)  
   
 return merged  
   
 def resolve\_duplicate\_names(self, name\_list: List[str]) -> str:  
 """  
 Given multiple names, pick the best one  
 Prefer: longer names, more specific names  
 """  
 if not name\_list:  
 return ""  
   
 if len(name\_list) == 1:  
 return name\_list[0]  
   
 # Remove duplicates  
 unique\_names = list(set(name\_list))  
   
 # Sort by length (longer is usually more specific)  
 unique\_names.sort(key=len, reverse=True)  
   
 return unique\_names[0]

## src\graph\graph\_builder.py

"""  
Graph Builder - CRITICAL FIX FOR UI DISPLAY  
Fixed query methods to return correct data structure  
"""  
  
import logging  
from typing import List, Dict, Optional  
from datetime import datetime  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class GraphBuilder:  
 def \_\_init\_\_(self, db):  
 self.db = db  
   
 # Division mapping  
 self.division\_map = {  
 'MSWIL': 'Wiring Systems',  
 'MSW': 'Wiring Systems',  
 'WIRING': 'Wiring Systems',  
 'SMR': 'Vision Systems',  
 'VISION': 'Vision Systems',  
 'SMP': 'Polymers',  
 'POLYMER': 'Polymers',  
 'SEATING': 'Seating Systems',  
 'LOGISTICS': 'Logistics',  
 'PKC': 'Wiring Systems'  
 }  
   
 # City to state mapping  
 self.city\_to\_state = {  
 'Sanand': 'Gujarat', 'Ahmedabad': 'Gujarat', 'Navagam': 'Gujarat',  
 'Pune': 'Maharashtra', 'Chakan': 'Maharashtra', 'Mumbai': 'Maharashtra',  
 'Chennai': 'Tamil Nadu', 'Hosur': 'Tamil Nadu',  
 'Bangalore': 'Karnataka', 'Bengaluru': 'Karnataka',  
 'Manesar': 'Haryana', 'Gurgaon': 'Haryana', 'Gurugram': 'Haryana',  
 'Noida': 'Uttar Pradesh', 'Haridwar': 'Uttarakhand',  
 'Bawal': 'Haryana', 'Dharuhera': 'Haryana'  
 }  
   
 logger.info("GraphBuilder initialized")  
   
 # ... (keep all existing methods: build\_graph, insert\_jobs, etc.)  
   
 # ✅ FIXED QUERY METHODS BELOW  
   
 def query\_facilities(self, division: Optional[str] = None, state: Optional[str] = None, status: Optional[str] = None) -> List[Dict]:  
 """  
 Return facilities with COMPLETE data structure for UI  
 ✅ FIXED: Now returns first\_date, url, and all required fields  
 """  
 sql = """  
 SELECT   
 f.id,  
 f.name,  
 f.city,  
 f.state,  
 f.country,  
 d.name AS division,  
 MIN(e.event\_date) AS first\_date, -- ✅ FIXED: Use MIN for first date  
 MAX(e.event\_date) AS last\_event\_date,  
 (SELECT status FROM events e2 WHERE e2.facility\_id = f.id ORDER BY e2.event\_date DESC LIMIT 1) AS status,  
 (SELECT expansion\_type FROM events e3 WHERE e3.facility\_id = f.id AND e3.expansion\_type IS NOT NULL LIMIT 1) AS expansion\_type,  
 (SELECT url FROM sources s JOIN evidence ev ON ev.source\_id = s.id WHERE ev.entity\_id = f.id AND ev.entity\_type = 'FACILITY' LIMIT 1) AS url,  
 (SELECT title FROM sources s JOIN evidence ev ON ev.source\_id = s.id WHERE ev.entity\_id = f.id AND ev.entity\_type = 'FACILITY' LIMIT 1) AS source\_title,  
 (SELECT publish\_date FROM sources s JOIN evidence ev ON ev.source\_id = s.id WHERE ev.entity\_id = f.id AND ev.entity\_type = 'FACILITY' LIMIT 1) AS publish\_date  
 FROM facilities f  
 LEFT JOIN divisions d ON f.division\_id = d.id  
 LEFT JOIN events e ON f.id = e.facility\_id  
 WHERE 1=1  
 """  
 params = []  
   
 if division:  
 sql += " AND d.name = ?"  
 params.append(division)  
 if state:  
 sql += " AND f.state = ?"  
 params.append(state)  
   
 sql += " GROUP BY f.id, f.name, f.city, f.state, f.country, d.name"  
   
 rows = self.db.execute\_query(sql, tuple(params))  
 results = []  
   
 for row in rows:  
 result = {  
 'id': row[0],  
 'name': row[1],  
 'facility': row[1], # ✅ Add both 'name' and 'facility' keys  
 'city': row[2],  
 'state': row[3],  
 'country': row[4],  
 'division': row[5],  
 'first\_date': row[6], # ✅ CRITICAL FIX  
 'last\_event\_date': row[7],  
 'status': row[8] or 'operational',  
 'expansion\_type': row[9],  
 'url': row[10] or 'https://www.motherson.com/contact/address-directory', # ✅ Default URL  
 'source\_title': row[11] or 'Motherson Address Directory',  
 'publish\_date': row[12],  
 'confidence': 0.9 # ✅ Add confidence score  
 }  
   
 # Apply status filter (after fetching)  
 if status:  
 if result.get('status', '').lower() == status.lower():  
 results.append(result)  
 else:  
 results.append(result)  
   
 logger.info(f"✅ query\_facilities returned {len(results)} facilities")  
 return results  
   
 def query\_expansions(self, date\_from: Optional[str] = None, date\_to: Optional[str] = None) -> List[Dict]:  
 """  
 Return expansion events with COMPLETE data structure  
 ✅ FIXED: Now includes timeline and better filtering  
 """  
 sql = """  
 SELECT   
 f.name AS facility,  
 f.city,  
 f.state,  
 d.name AS division,  
 e.event\_date,  
 e.expansion\_type,  
 e.status,  
 e.confidence,  
 s.url,  
 s.title AS source\_title,  
 s.publish\_date  
 FROM events e  
 JOIN facilities f ON e.facility\_id = f.id  
 LEFT JOIN divisions d ON f.division\_id = d.id  
 LEFT JOIN evidence ev ON ev.entity\_id = f.id AND ev.entity\_type = 'FACILITY'  
 LEFT JOIN sources s ON ev.source\_id = s.id  
 WHERE 1=1  
 """  
   
 params = []  
   
 # ✅ FIXED: Broaden the query to include more results  
 # Look for ANY event with a date in range, not just expansion\_type  
 if date\_from:  
 sql += " AND e.event\_date >= ?"  
 params.append(date\_from)  
 if date\_to:  
 sql += " AND e.event\_date <= ?"  
 params.append(date\_to)  
   
 sql += " AND (e.expansion\_type IS NOT NULL OR e.status IN ('planned', 'under-construction'))"  
 sql += " ORDER BY e.event\_date DESC"  
   
 rows = self.db.execute\_query(sql, tuple(params))  
 results = []  
   
 for r in rows:  
 result = {  
 'facility': r[0],  
 'city': r[1],  
 'state': r[2],  
 'division': r[3],  
 'event\_date': r[4],  
 'timeline': r[4], # ✅ Add timeline field  
 'expansion\_type': r[5] or 'expansion', # ✅ Default value  
 'status': r[6],  
 'confidence': r[7] or 0.7,  
 'url': r[8] or 'https://www.motherson.com',  
 'source\_title': r[9] or 'Motherson Document',  
 'publish\_date': r[10]  
 }  
 results.append(result)  
   
 logger.info(f"✅ query\_expansions returned {len(results)} expansions")  
 return results  
   
 def query\_jobs(self, factory\_only: bool = False) -> List[Dict]:  
 """  
 Return job postings with COMPLETE data structure  
 ✅ FIXED: Now includes proper URL and location mapping  
 """  
 sql = """  
 SELECT   
 j.id,  
 j.title,  
 j.location,  
 j.posted\_date,  
 j.description,  
 f.name AS facility,  
 d.name AS division,  
 j.is\_factory\_role,  
 s.url,  
 s.title AS source\_title  
 FROM jobs j  
 LEFT JOIN facilities f ON j.facility\_id = f.id  
 LEFT JOIN divisions d ON j.division\_id = d.id  
 LEFT JOIN sources s ON j.source\_id = s.id  
 WHERE 1=1  
 """  
   
 params = []  
 if factory\_only:  
 sql += " AND j.is\_factory\_role = 1"  
   
 sql += " ORDER BY j.posted\_date DESC NULLS LAST"  
   
 rows = self.db.execute\_query(sql, tuple(params))  
 results = []  
   
 for r in rows:  
 result = {  
 'id': r[0],  
 'title': r[1],  
 'location': r[2] or 'India',  
 'posted\_date': r[3],  
 'description': r[4],  
 'facility': r[5] or 'Multiple Locations',  
 'division': r[6] or 'Unknown',  
 'is\_factory\_role': bool(r[7]),  
 'url': r[8] or 'https://careers.motherson.com',  
 'source\_title': r[9] or 'Motherson Careers',  
 'confidence': 0.85 # ✅ Add confidence  
 }  
 results.append(result)  
   
 logger.info(f"✅ query\_jobs returned {len(results)} jobs")  
 return results  
   
 # ... (keep all other existing methods unchanged)

## src\graph\\_\_init\_\_.py

# ==================================================  
# File: src/graph/\_\_init\_\_.py  
# ==================================================  
"""  
Knowledge graph and entity resolution module  
"""  
  
from .database import Database  
from .entity\_resolver import EntityResolver  
from .graph\_builder import GraphBuilder  
  
\_\_all\_\_ = ['Database', 'EntityResolver', 'GraphBuilder']

## src\ner\entity\_extractor.py

"""  
Enhanced Entity Extractor - COMPLETE REWRITE  
Significantly improved facility, division, and job extraction  
"""  
  
import spacy  
import re  
import logging  
from typing import Dict, List  
import datetime  
  
try:  
 from src.config import SPACY\_MODEL  
except ImportError:  
 SPACY\_MODEL = "en\_core\_web\_sm"  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class EntityExtractor:  
 def \_\_init\_\_(self):  
 try:  
 self.nlp = spacy.load(SPACY\_MODEL)  
 logger.info(f"Loaded spaCy model: {SPACY\_MODEL}")  
 except:  
 logger.warning("spaCy model not found. Run: python -m spacy download en\_core\_web\_sm")  
 self.nlp = None  
   
 # EXPANDED Division patterns  
 self.division\_patterns = [  
 r'\b(Motherson Wiring Systems?|Wiring Systems?|Wiring Division|MSWIL|MSW)\b',  
 r'\b(Motherson Vision Systems?|Vision Systems?|Vision Division|SMR|Samvardhana Motherson Reflectec)\b',  
 r'\b(Motherson Seating Systems?|Seating Systems?|Seating Division)\b',  
 r'\b(Motherson Polymers?|Polymers Division|SMP)\b',  
 r'\b(Motherson Logistics|Logistics Division)\b',  
 r'\b(PKC Group|PKC Wiring)\b',  
 ]  
   
 # Division mapping  
 self.division\_map = {  
 'MSWIL': 'Wiring Systems',  
 'MSW': 'Wiring Systems',  
 'WIRING': 'Wiring Systems',  
 'SMR': 'Vision Systems',  
 'VISION': 'Vision Systems',  
 'SMP': 'Polymers',  
 'POLYMER': 'Polymers',  
 'SEATING': 'Seating Systems',  
 'LOGISTICS': 'Logistics',  
 'PKC': 'Wiring Systems'  
 }  
   
 # Status patterns  
 self.status\_patterns = {  
 'operational': r'\b(operational|operating|commissioned|inaugurated|started operations|in operation|existing plant|currently operating|active)\b',  
 'under-construction': r'\b(under construction|being built|construction phase|currently building|construction underway|construction started|under development)\b',  
 'planned': r'\b(planned|proposed|announced|upcoming|future|will establish|to be set up|proposed plant|plans to set up|announced plant)\b'  
 }  
   
 # EXPANDED Event patterns  
 self.event\_patterns = [  
 r'\b(announced?|announcing|announcement of)\b',  
 r'\b(groundbreaking|foundation stone|ground breaking)\b',  
 r'\b(commissioned?|commissioning|inaugurated?|inauguration)\b',  
 r'\b(expansion|expanding|expand|phase \d+|scale up|brownfield)\b',  
 r'\b(greenfield|new plant|new facility|new manufacturing|new unit)\b',  
 r'\b(set up|setting up|establish|establishing|established)\b',  
 r'\b(started operations?|commenced operations?|begin operations?)\b',  
 ]  
   
 # EXPANDED Date patterns  
 self.date\_patterns = [  
 r'\b(FY\s\*\d{2,4}(?:-\d{2,4})?)\b',  
 r'\b(Q[1-4]\s+\d{4})\b',  
 r'\b(January|February|March|April|May|June|July|August|September|October|November|December)\s+(\d{4})\b',  
 r'\b(\d{1,2})[-/](\d{1,2})[-/](\d{4})\b',  
 r'\b(202[0-9]|201[5-9])\b',  
 r'\b(H[1-2]\s+(?:FY\s\*)?\d{4})\b', # Half-year format  
 ]  
   
 # EXPANDED Indian locations  
 self.indian\_states = [  
 'Gujarat', 'Tamil Nadu', 'Maharashtra', 'Haryana', 'Karnataka',  
 'Uttar Pradesh', 'Rajasthan', 'Punjab', 'Telangana', 'Andhra Pradesh',  
 'West Bengal', 'Madhya Pradesh', 'Kerala', 'Odisha', 'Uttarakhand'  
 ]  
   
 self.indian\_cities = [  
 'Ahmedabad', 'Pune', 'Chennai', 'Bangalore', 'Bengaluru', 'Mumbai',  
 'Gurgaon', 'Gurugram', 'Hyderabad', 'Kolkata', 'Sanand', 'Navagam',  
 'Chakan', 'Manesar', 'Noida', 'Haridwar', 'Bawal', 'Hosur',  
 'Dharuhera', 'Greater Noida', 'Aurangabad', 'Coimbatore'  
 ]  
   
 # City to state mapping  
 self.city\_to\_state = {  
 'Sanand': 'Gujarat', 'Ahmedabad': 'Gujarat', 'Navagam': 'Gujarat',  
 'Pune': 'Maharashtra', 'Chakan': 'Maharashtra', 'Mumbai': 'Maharashtra', 'Aurangabad': 'Maharashtra',  
 'Chennai': 'Tamil Nadu', 'Hosur': 'Tamil Nadu', 'Coimbatore': 'Tamil Nadu',  
 'Bangalore': 'Karnataka', 'Bengaluru': 'Karnataka',  
 'Manesar': 'Haryana', 'Gurgaon': 'Haryana', 'Gurugram': 'Haryana', 'Bawal': 'Haryana', 'Dharuhera': 'Haryana',  
 'Noida': 'Uttar Pradesh', 'Greater Noida': 'Uttar Pradesh',  
 'Haridwar': 'Uttarakhand',  
 'Hyderabad': 'Telangana',  
 'Kolkata': 'West Bengal'  
 }  
   
 # STRICTER facility blacklist  
 self.facility\_blacklist = [  
 r'\b(board of directors|management team|audit committee)\b',  
 r'\b(page \d+|section \d+|chapter \d+)\b',  
 r'\b(registered office|corporate office|head office)\b',  
 r'\b(financial statement|balance sheet|profit and loss)\b',  
 r'\b(total assets|total revenue|net profit)\b',  
 r'^(the|a|an|this|that|our|their)\b',  
 r'\b(company|corporation|limited|group)$',  
 ]  
   
 # EXPANDED Factory job keywords  
 self.factory\_job\_keywords = [  
 'plant', 'production', 'manufacturing', 'assembly', 'operator',  
 'technician', 'mechanic', 'maintenance', 'quality', 'supervisor',  
 'foreman', 'machinist', 'welder', 'fitter', 'shift', 'floor',  
 'warehouse', 'logistics', 'supply chain', 'material', 'inventory',  
 'tool', 'die', 'mold', 'press', 'injection', 'stamping',  
 'paint', 'welding', 'inspection', 'process', 'line'  
 ]  
   
 self.non\_factory\_keywords = [  
 'software', 'developer', 'programmer', 'data scientist',  
 'it ', 'digital', 'cyber', 'application', 'web', 'mobile',  
 'cloud', 'devops', 'analyst', 'sap', 'erp'  
 ]  
   
 def extract\_facilities(self, text: str) -> List[Dict]:  
 """Extract facility mentions - COMPREHENSIVE PATTERNS"""  
 facilities = []  
   
 # Pattern 1: "City Plant/Facility/Unit"  
 pattern1 = r'\b(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida|Gurgaon|Gurugram|Hyderabad|Mumbai|Dharuhera|Coimbatore|Aurangabad|Greater Noida)\s+(Plant|Facility|Unit|Manufacturing|Operations|Factory|Site|Campus)\b'  
   
 # Pattern 2: "facility in City" or "plant at City"  
 pattern2 = r'\b(plant|facility|unit|manufacturing|operations)\s+(?:in|at|located in|located at)\s+(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida|Gurgaon|Gurugram|Hyderabad|Mumbai|Dharuhera)\b'  
   
 # Pattern 3: "Division City" (e.g., "Wiring Sanand")  
 pattern3 = r'\b(Wiring|Vision|Seating|Polymers|Logistics)\s+(?:Systems?\s+)?(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida)\b'  
   
 # Pattern 4: Context-based  
 pattern4 = r'(?:announced|established|set up|commissioned|operates|inaugurated?)(?:\s+\w+){0,8}\s+(?:plant|facility|unit|manufacturing)?\s\*(?:in|at)\s+(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida|Gurgaon|Gurugram|Hyderabad|Mumbai)\b'  
   
 # Pattern 5: "MSWIL/SMR/SMP City"  
 pattern5 = r'\b(MSWIL|SMR|SMP|PKC)\s+(?:plant|facility|unit)?\s\*(?:at|in)?\s\*(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida)\b'  
   
 # Pattern 6: "City, State" format  
 pattern6 = r'\b(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida|Gurgaon|Gurugram|Hyderabad|Mumbai),\s\*(Gujarat|Tamil Nadu|Maharashtra|Haryana|Karnataka|Uttar Pradesh|Uttarakhand|Telangana)\b'  
   
 # Pattern 7: Table-like patterns  
 pattern7 = r'(?:plant|facility|unit|manufacturing|site)\s\*[:\-]\s\*(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida)\b'  
   
 all\_patterns = [pattern1, pattern2, pattern3, pattern4, pattern5, pattern6, pattern7]  
   
 for pattern in all\_patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 matched\_text = match.group(0)  
   
 # Extract city  
 city = None  
 for city\_name in self.indian\_cities:  
 if city\_name.lower() in matched\_text.lower():  
 city = city\_name  
 break  
   
 if not city:  
 continue  
   
 # Build facility name  
 facility\_name = None  
   
 if pattern == pattern3:  
 division = match.group(1)  
 facility\_name = f"{division} {city} Plant"  
 elif pattern == pattern5:  
 prefix = match.group(1)  
 facility\_name = f"{prefix} {city} Plant"  
 elif 'MSWIL' in matched\_text.upper():  
 facility\_name = f"MSWIL {city} Plant"  
 elif 'SMR' in matched\_text.upper():  
 facility\_name = f"SMR {city} Plant"  
 elif 'SMP' in matched\_text.upper():  
 facility\_name = f"SMP {city} Plant"  
 else:  
 facility\_name = f"{city} Plant"  
   
 if self.\_is\_valid\_facility\_name(facility\_name):  
 facilities.append({  
 'text': facility\_name,  
 'label': 'FACILITY',  
 'start': match.start(),  
 'end': match.end()  
 })  
   
 # Deduplicate  
 unique\_facilities = {}  
 for fac in facilities:  
 if fac['text'] not in unique\_facilities:  
 unique\_facilities[fac['text']] = fac  
   
 return list(unique\_facilities.values())  
   
 def \_is\_valid\_facility\_name(self, name: str) -> bool:  
 """Check if facility name is valid"""  
 name\_lower = name.lower()  
   
 if len(name) < 5 or len(name) > 100:  
 return False  
   
 if re.match(r'^\d+', name):  
 return False  
   
 for pattern in self.facility\_blacklist:  
 if re.search(pattern, name\_lower):  
 return False  
   
 return True  
   
 def extract\_divisions(self, text: str) -> List[Dict]:  
 """Extract divisions"""  
 divisions = []  
   
 for pattern in self.division\_patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 div\_name = match.group(0).strip()  
   
 # Normalize  
 div\_upper = div\_name.upper()  
 normalized\_name = None  
   
 for abbr, full\_name in self.division\_map.items():  
 if abbr in div\_upper:  
 normalized\_name = full\_name  
 break  
   
 if not normalized\_name:  
 if 'wiring' in div\_name.lower():  
 normalized\_name = 'Wiring Systems'  
 elif 'vision' in div\_name.lower():  
 normalized\_name = 'Vision Systems'  
 elif 'seating' in div\_name.lower():  
 normalized\_name = 'Seating Systems'  
 elif 'polymer' in div\_name.lower():  
 normalized\_name = 'Polymers'  
 elif 'logistic' in div\_name.lower():  
 normalized\_name = 'Logistics'  
 else:  
 normalized\_name = div\_name  
   
 divisions.append({  
 'text': normalized\_name,  
 'label': 'DIVISION',  
 'start': match.start(),  
 'end': match.end()  
 })  
   
 # Deduplicate  
 unique\_divisions = {}  
 for div in divisions:  
 if div['text'] not in unique\_divisions:  
 unique\_divisions[div['text']] = div  
   
 return list(unique\_divisions.values())  
   
 def extract\_locations(self, text: str) -> List[Dict]:  
 """Extract locations"""  
 locations = []  
   
 # Extract states  
 for state in self.indian\_states:  
 for match in re.finditer(r'\b' + re.escape(state) + r'\b', text):  
 locations.append({  
 'text': state,  
 'label': 'LOCATION',  
 'start': match.start(),  
 'end': match.end()  
 })  
   
 # Extract cities  
 for city in self.indian\_cities:  
 for match in re.finditer(r'\b' + re.escape(city) + r'\b', text):  
 locations.append({  
 'text': city,  
 'label': 'LOCATION',  
 'start': match.start(),  
 'end': match.end()  
 })  
   
 # City, State pattern  
 city\_state\_pattern = r'\b([A-Z][a-z]{2,}),\s\*([A-Z][a-z]+(?:\s+[A-Z][a-z]+)?)\b'  
 for match in re.finditer(city\_state\_pattern, text):  
 city = match.group(1).strip()  
 state = match.group(2).strip()  
   
 if city in self.indian\_cities or state in self.indian\_states:  
 locations.append({  
 'text': match.group(0).strip(),  
 'label': 'LOCATION',  
 'start': match.start(),  
 'end': match.end()  
 })  
   
 # Deduplicate  
 unique\_locations = {}  
 for loc in locations:  
 if loc['text'] not in unique\_locations:  
 unique\_locations[loc['text']] = loc  
   
 return list(unique\_locations.values())  
   
 def extract\_status(self, text: str) -> List[Dict]:  
 """Extract status mentions"""  
 statuses = []  
 for status\_type, pattern in self.status\_patterns.items():  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 statuses.append({  
 'text': status\_type,  
 'label': 'STATUS',  
 'start': match.start(),  
 'end': match.end()  
 })  
 return statuses  
   
 def extract\_events(self, text: str) -> List[Dict]:  
 """Extract event mentions"""  
 events = []  
 for pattern in self.event\_patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 events.append({  
 'text': match.group(0).strip(),  
 'label': 'EVENT',  
 'start': match.start(),  
 'end': match.end()  
 })  
 return events  
   
 def extract\_dates(self, text: str) -> List[Dict]:  
 """Extract and normalize dates"""  
 dates = []  
   
 for pattern in self.date\_patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 date\_text = match.group(0).strip()  
 normalized\_date = self.\_normalize\_date(date\_text)  
   
 if normalized\_date:  
 dates.append({  
 'text': normalized\_date,  
 'label': 'DATE',  
 'start': match.start(),  
 'end': match.end()  
 })  
   
 # Deduplicate  
 unique\_dates = {}  
 for date in dates:  
 if date['text'] not in unique\_dates:  
 unique\_dates[date['text']] = date  
   
 return list(unique\_dates.values())  
   
 def \_normalize\_date(self, date\_text: str) -> str:  
 """Normalize date to ISO format"""  
 date\_text = date\_text.strip()  
   
 try:  
 # FY format  
 if date\_text.upper().startswith('FY'):  
 year\_match = re.search(r'\d{4}', date\_text)  
 if year\_match:  
 year = year\_match.group(0)  
 return f"{year}-03-31"  
 return None  
 except Exception as e:  
 logger.warning(f"Error processing FY format for '{date\_text}': {e}")  
 return None  
   
 # Quarter format  
 quarter\_match = re.match(r'Q([1-4])\s+(\d{4})', date\_text, re.IGNORECASE)  
 if quarter\_match:  
 quarter = int(quarter\_match.group(1))  
 year = quarter\_match.group(2)  
 month = quarter \* 3  
 return f"{year}-{month:02d}-01"  
   
 # Half-year format  
 half\_match = re.match(r'H([1-2])\s+(?:FY\s\*)?(\d{4})', date\_text, re.IGNORECASE)  
 if half\_match:  
 half = int(half\_match.group(1))  
 year = half\_match.group(2)  
 month = 6 if half == 1 else 12  
 return f"{year}-{month:02d}-01"  
   
 # Month Year format  
 month\_year = re.match(r'(January|February|March|April|May|June|July|August|September|October|November|December)\s+(\d{4})', date\_text, re.IGNORECASE)  
 if month\_year:  
 months = ['January', 'February', 'March', 'April', 'May', 'June',  
 'July', 'August', 'September', 'October', 'November', 'December']  
 month = months.index(month\_year.group(1).capitalize()) + 1  
 year = month\_year.group(2)  
 return f"{year}-{month:02d}-01"  
   
 # Year only  
 if re.match(r'^\d{4}', date\_text):  
 year = int(date\_text)  
 if 2010 <= year <= datetime.datetime.now().year + 10:  
 return f"{date\_text}-01-01"  
   
 return None  
   
 except Exception as e:  
 logger.warning(f"Date normalization failed for '{date\_text}': {e}")  
 return None  
   
 def extract\_job\_titles(self, text: str) -> List[Dict]:  
 """Extract job titles with locations - ENHANCED VERSION"""  
 jobs = []  
   
 # ENHANCED job patterns WITH location capture  
 job\_patterns = [  
 # Pattern: "Title - Location" or "Title | Location"  
 r'\b(Plant|Production|Manufacturing|Assembly|Quality|Maintenance|Warehouse|Logistics|Tool|Die|Mold|Process|Line)\s+(Manager|Engineer|Supervisor|Operator|Technician|Coordinator|Specialist|Planner|Designer|Controller|Lead|Head|In-charge)(?:\s\*[-|]\s\*|\s+at\s+|\s+in\s+|\s+-\s+)?(Chennai|Pune|Bangalore|Bengaluru|Hosur|Sanand|Manesar|Ahmedabad|Mumbai|Hyderabad|Tamil Nadu|Gujarat|Maharashtra|Haryana|Karnataka|India)?\b',  
   
 # Pattern: "Senior/Junior Title" with location  
 r'\b(Senior|Junior|Lead|Chief|Deputy|Assistant|Sr\.|Jr\.)\s+\w+\s+(Engineer|Manager|Supervisor|Coordinator|Technician|Specialist)(?:\s\*[-|]\s\*|\s+at\s+|\s+in\s+|\s+-\s+)?(Chennai|Pune|Bangalore|Bengaluru|Hosur|Sanand|Manesar|Ahmedabad|Mumbai|Hyderabad)?\b',  
   
 # Pattern: "Role Technician/Operator"  
 r'\b\w+\s+(Operator|Technician|Mechanic|Fitter|Welder|Assembler|Inspector|Machinist)(?:\s\*[-|]\s\*|\s+at\s+|\s+in\s+|\s+-\s+)?(Chennai|Pune|Bangalore|Bengaluru|Hosur|Sanand|Manesar|Ahmedabad)?\b',  
   
 # Pattern: "Shift/Floor Manager"  
 r'\b(Shift|Floor|Line|Production|Process|Material)\s+(Manager|Supervisor|Coordinator|In-charge|Lead|Engineer)(?:\s\*[-|]\s\*|\s+at\s+|\s+in\s+|\s+-\s+)?(Chennai|Pune|Bangalore|Bengaluru|Hosur|Sanand|Manesar)?\b',  
   
 # Pattern: Additional factory roles  
 r'\b(Inventory|Supply Chain|Stamping|Welding|Painting|Injection|Molding)\s+(Engineer|Manager|Supervisor|Technician|Specialist)(?:\s\*[-|]\s\*|\s+at\s+|\s+in\s+)?(Chennai|Pune|Bangalore|Bengaluru|Hosur|Sanand)?\b',  
 ]  
   
 for pattern in job\_patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 # Extract title  
 if match.lastindex and match.lastindex >= 2:  
 title\_parts = [match.group(i) for i in range(1, min(match.lastindex, 3))]  
 title = ' '.join(p for p in title\_parts if p).strip()  
 else:  
 title = match.group(0).strip()  
   
 # Extract location if captured  
 location = None  
 if match.lastindex and match.lastindex >= 3 and match.group(match.lastindex):  
 location = match.group(match.lastindex).strip()  
   
 # Search for location in nearby context if not captured  
 if not location:  
 context\_end = min(match.end() + 200, len(text))  
 context = text[match.end():context\_end]  
   
 # Look for city names  
 for city in self.indian\_cities:  
 if city in context:  
 location = city  
 break  
   
 # Look for state names  
 if not location:  
 for state in self.indian\_states:  
 if state in context:  
 location = state  
 break  
   
 title\_lower = title.lower()  
   
 # Check if it's a factory role  
 is\_factory = any(kw in title\_lower for kw in self.factory\_job\_keywords)  
 is\_non\_factory = any(kw in title\_lower for kw in self.non\_factory\_keywords)  
   
 if is\_factory and not is\_non\_factory:  
 jobs.append({  
 'text': title,  
 'location': location or 'India',  
 'label': 'JOB\_TITLE',  
 'start': match.start(),  
 'end': match.end(),  
 'is\_factory\_role': True  
 })  
   
 return jobs  
   
 def extract\_entities(self, text: str) -> Dict[str, List[Dict]]:  
 """Extract all entities"""  
 if not self.nlp:  
 logger.error("NLP model not loaded")  
 return {}  
   
 logger.info(f"Extracting entities from {len(text)} chars")  
   
 entities = {  
 'facilities': self.extract\_facilities(text),  
 'divisions': self.extract\_divisions(text),  
 'locations': self.extract\_locations(text),  
 'status': self.extract\_status(text),  
 'events': self.extract\_events(text),  
 'dates': self.extract\_dates(text),  
 'job\_titles': self.extract\_job\_titles(text)  
 }  
   
 total = sum(len(v) for v in entities.values())  
 logger.info(f"Extracted {total} entities total")  
 return entities # Ensure all entities are returned properly

## src\ner\\_\_init\_\_.py

# ==================================================  
# File: src/ner/\_\_init\_\_.py  
# ==================================================  
"""  
Named Entity Recognition and extraction module  
"""  
  
from .entity\_extractor import EntityExtractor  
  
\_\_all\_\_ = ['EntityExtractor']

## src\rag\generator.py

"""  
Generator - FINAL FIX  
Correct Gemini model name + better fallback handling  
"""  
  
import logging  
from typing import List, Dict, Optional  
import google.generativeai as genai  
  
from src.config import GEMINI\_API\_KEY  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
class Generator:  
 def \_\_init\_\_(self, api\_key: str = None):  
 api\_key = api\_key or GEMINI\_API\_KEY  
   
 if api\_key:  
 try:  
 genai.configure(api\_key=api\_key)  
 # CRITICAL FIX: Use correct model name for API v1  
 self.model = genai.GenerativeModel('gemini-1.5-flash-002')  
 logger.info("Gemini API initialized with gemini-1.5-flash-002")  
 except Exception as e:  
 logger.warning(f"Gemini initialization failed: {e}. Using fallback mode.")  
 self.model = None  
 else:  
 self.model = None  
 logger.warning("No Gemini API key. LLM features disabled")  
   
 def generate\_answer(  
 self,  
 query: str,  
 graph\_results: List[Dict],  
 vector\_passages: List[Dict] = None  
 ) -> str:  
 """Generate evidence-backed answer with fallback"""  
 if not graph\_results:  
 return "❌ No evidence found in the database."  
   
 # Try LLM if available  
 if self.model:  
 try:  
 context = self.\_build\_context(graph\_results, vector\_passages)  
 prompt = self.\_build\_prompt(query, context, len(graph\_results))  
 response = self.model.generate\_content(prompt)  
 return response.text  
 except Exception as e:  
 logger.error(f"LLM generation error: {e}")  
 logger.info("Falling back to simple answer generation")  
   
 # Fallback to simple answer  
 return self.\_generate\_fallback\_answer(graph\_results)  
   
 def \_build\_context(self, graph\_results: List[Dict], vector\_passages: List[Dict] = None) -> str:  
 """Build evidence context for LLM"""  
 context = "# Evidence from Database:\n\n"  
   
 for idx, result in enumerate(graph\_results[:20], 1):  
 context += f"[{idx}] "  
   
 if 'facility' in result or 'name' in result:  
 facility\_name = result.get('facility') or result.get('name', 'N/A')  
 context += f"Facility: {facility\_name}\n"  
 context += f" Division: {result.get('division', 'N/A')}\n"  
 context += f" Location: {result.get('city', 'N/A')}, {result.get('state', 'N/A')}\n"  
 context += f" Status: {result.get('status', 'N/A')}\n"  
   
 if result.get('expansion\_type'):  
 context += f" Type: {result['expansion\_type']}\n"  
   
 if result.get('last\_event\_date') or result.get('event\_date'):  
 date = result.get('last\_event\_date') or result.get('event\_date')  
 context += f" Date: {date}\n"  
   
 if result.get('url'):  
 context += f" Source: {result['url']}\n"  
   
 elif 'title' in result:  
 context += f"Job: {result.get('title', 'N/A')}\n"  
 context += f" Location: {result.get('location', 'N/A')}\n"  
 context += f" Facility: {result.get('facility', 'N/A')}\n"  
 if result.get('url'):  
 context += f" Source: {result['url']}\n"  
   
 context += "\n"  
   
 if vector\_passages:  
 context += "\n# Supporting Document Passages:\n\n"  
 for idx, passage in enumerate(vector\_passages[:3], 1):  
 context += f"[P{idx}] {passage['text'][:300]}...\n"  
 context += f" Source: {passage['metadata'].get('url', 'N/A')}\n\n"  
   
 return context  
   
 def \_build\_prompt(self, query: str, context: str, evidence\_count: int) -> str:  
 """Build prompt with strict guardrails"""  
 return f"""You are a precise analyst providing evidence-based answers about Motherson India facilities.  
  
STRICT RULES:  
1. Answer ONLY using the evidence provided below  
2. Every claim MUST be supported by a citation [1], [2], etc.  
3. If evidence is weak, explicitly state this  
4. DO NOT invent information  
5. Be concise and factual  
  
Evidence Available: {evidence\_count} records  
  
{context}  
  
User Query: {query}  
  
Answer with bullet points for facilities/jobs, using citations [1], [2] for each claim:"""  
   
 def \_generate\_fallback\_answer(self, graph\_results: List[Dict]) -> str:  
 """Generate simple answer without LLM"""  
 if not graph\_results:  
 return "No results found in the database."  
   
 answer = f"Found {len(graph\_results)} results:\n\n"  
   
 for idx, result in enumerate(graph\_results[:10], 1):  
 if 'facility' in result or 'name' in result:  
 facility\_name = result.get('facility') or result.get('name', 'N/A')  
 answer += f"{idx}. {facility\_name} "  
 answer += f"({result.get('division', 'N/A')}) - "  
 answer += f"{result.get('city', 'N/A')}, {result.get('state', 'N/A')}\n"  
 elif 'title' in result:  
 answer += f"{idx}. {result.get('title', 'N/A')} - "  
 answer += f"{result.get('location', 'N/A')}\n"  
   
 if len(graph\_results) > 10:  
 answer += f"\n...and {len(graph\_results) - 10} more results"  
   
 return answer  
   
 def apply\_guardrails(self, results: List[Dict], query\_type: str) -> Dict:  
 """Apply guardrails and quality checks"""  
 if not results:  
 return {  
 'data': [],  
 'evidence': [],  
 'warning': '⚠️ No evidence found. Results may be incomplete.',  
 'confidence': 0.0,  
 'answer': None  
 }  
   
 # Calculate confidence  
 confidences = [r.get('confidence', 0.7) for r in results]  
 avg\_confidence = sum(confidences) / len(confidences) if confidences else 0.7  
   
 # Determine warning  
 warning = None  
 if avg\_confidence < 0.5:  
 warning = "⚠️ Low confidence results. Evidence may be weak."  
 elif avg\_confidence < 0.7:  
 warning = "⚠️ Moderate confidence. Some results may need verification."  
   
 # Filter for hiring query  
 if query\_type == 'hiring\_positions':  
 non\_factory\_keywords = ['software', 'developer', 'programmer', 'data scientist', 'analyst', 'it ', 'digital']  
 filtered\_results = []  
   
 for result in results:  
 title = result.get('title', '').lower()  
 is\_non\_factory = any(kw in title for kw in non\_factory\_keywords)  
   
 if not is\_non\_factory or result.get('is\_factory\_role'):  
 filtered\_results.append(result)  
   
 removed\_count = len(results) - len(filtered\_results)  
 if removed\_count > 0:  
 warning = f"ℹ️ Filtered out {removed\_count} non-factory roles"  
   
 results = filtered\_results  
   
 # Build evidence list  
 evidence = self.\_build\_evidence\_list(results)  
   
 return {  
 'data': results,  
 'evidence': evidence,  
 'warning': warning,  
 'confidence': avg\_confidence,  
 'answer': None  
 }  
   
 def \_build\_evidence\_list(self, results: List[Dict]) -> List[Dict]:  
 """Build evidence list with highlighting info"""  
 evidence = []  
   
 for idx, result in enumerate(results):  
 if 'facility' in result or 'name' in result:  
 facility\_name = result.get('facility') or result.get('name', 'N/A')  
 snippet = f"Facility: {facility\_name}\n"  
 snippet += f"Division: {result.get('division', 'N/A')}\n"  
 snippet += f"Location: {result.get('city', 'N/A')}, {result.get('state', 'N/A')}\n"  
 snippet += f"Status: {result.get('status', 'N/A')}"  
   
 if result.get('expansion\_type'):  
 snippet += f"\nType: {result['expansion\_type']}"  
   
 elif 'title' in result:  
 snippet = f"Job Title: {result.get('title', 'N/A')}\n"  
 snippet += f"Location: {result.get('location', 'N/A')}\n"  
 snippet += f"Facility: {result.get('facility', 'N/A')}"  
   
 else:  
 snippet = result.get('text\_snippet', 'N/A')  
   
 evidence.append({  
 'id': idx,  
 'title': result.get('source\_title', f"Evidence {idx + 1}"),  
 'text': snippet,  
 'url': result.get('url', 'N/A'),  
 'date': result.get('publish\_date') or result.get('last\_event\_date') or result.get('event\_date') or result.get('posted\_date') or 'N/A',  
 'source\_type': result.get('source\_type', 'Document'),  
 'char\_start': result.get('char\_start'),  
 'char\_end': result.get('char\_end'),  
 'confidence': result.get('confidence', 0.7)  
 })  
   
 return evidence

## src\rag\query\_classifier.py

"""  
Query Classifier - Classifies user queries into 3 core tasks  
"""  
  
import logging  
from typing import Dict  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class QueryClassifier:  
 def \_\_init\_\_(self):  
 # Keywords for each query type  
 self.query\_keywords = {  
 'list\_facilities': [  
 'list', 'show', 'display', 'all facilities', 'facilities by division',  
 'what facilities', 'which facilities', 'facilities in', 'plants in'  
 ],  
 'new\_expansions': [  
 'new', 'expansion', 'greenfield', 'brownfield', 'expanded',  
 'recent', 'latest', 'upcoming', 'announced', 'phase',  
 'new plant', 'new facility', 'expansion plan', 'future plant'  
 ],  
 'hiring\_positions': [  
 'hiring', 'jobs', 'positions', 'careers', 'recruitment',  
 'vacancies', 'openings', 'job postings', 'employment',  
 'looking for', 'hiring for', 'factory roles'  
 ]  
 }  
   
 def classify(self, query: str) -> str:  
 """  
 Classify query into one of 3 types  
 Returns: 'list\_facilities', 'new\_expansions', or 'hiring\_positions'  
 """  
 if not query:  
 return 'list\_facilities'  
   
 query\_lower = query.lower()  
   
 # Calculate scores for each query type  
 scores = {}  
   
 for query\_type, keywords in self.query\_keywords.items():  
 score = 0  
 for keyword in keywords:  
 if keyword in query\_lower:  
 # Longer keyword matches get higher scores  
 score += len(keyword.split())  
 scores[query\_type] = score  
   
 # Return type with highest score  
 if max(scores.values()) == 0:  
 # No clear match, default to list\_facilities  
 logger.info(f"No clear classification for '{query}', defaulting to list\_facilities")  
 return 'list\_facilities'  
   
 classified\_type = max(scores, key=scores.get)  
 logger.info(f"Classified query '{query}' as: {classified\_type}")  
   
 return classified\_type  
   
 def get\_query\_description(self, query\_type: str) -> str:  
 """Get human-readable description of query type"""  
 descriptions = {  
 'list\_facilities': 'List all Motherson facilities by division',  
 'new\_expansions': 'Show new/expanded plants in the last 24 months',  
 'hiring\_positions': 'Surface hiring positions for factory roles'  
 }  
 return descriptions.get(query\_type, 'Unknown query type')

## src\rag\retriever.py

# ==================================================  
# File: src/rag/retriever.py  
# ==================================================  
  
import logging  
from typing import List, Dict, Optional  
import chromadb  
from chromadb.config import Settings  
from pathlib import Path  
from datetime import datetime, timedelta  
  
from ..config import VECTOR\_STORE\_DIR, CHROMA\_COLLECTION, CHUNK\_SIZE, CHUNK\_OVERLAP  
from src.graph.database import Database  
from src.graph.graph\_builder import GraphBuilder  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
class Retriever:  
 def \_\_init\_\_(self):  
 # Graph setup  
 db = Database()  
 self.graph\_builder = GraphBuilder(db)  
  
 # ChromaDB setup  
 self.chroma\_client = chromadb.PersistentClient(  
 path=str(VECTOR\_STORE\_DIR),  
 settings=Settings(anonymized\_telemetry=False)  
 )  
  
 try:  
 self.collection = self.chroma\_client.get\_collection(CHROMA\_COLLECTION)  
 logger.info(f"Loaded existing collection: {CHROMA\_COLLECTION}")  
 except:  
 self.collection = self.chroma\_client.create\_collection(  
 name=CHROMA\_COLLECTION,  
 metadata={"description": "Motherson documents"}  
 )  
 logger.info(f"Created new collection: {CHROMA\_COLLECTION}")  
  
 def \_clean\_metadata(self, metadata: Dict) -> Dict:  
 cleaned = {}  
 for key, value in metadata.items():  
 if value is None:  
 cleaned[key] = ''  
 elif isinstance(value, (str, int, float, bool)):  
 cleaned[key] = value  
 else:  
 cleaned[key] = str(value)  
 return cleaned  
  
 def index\_documents(self, documents: List[Dict]):  
 try:  
 self.chroma\_client.delete\_collection(CHROMA\_COLLECTION)  
 self.collection = self.chroma\_client.create\_collection(CHROMA\_COLLECTION)  
 logger.info("Cleared existing collection")  
 except Exception as e:  
 logger.warning(f"Could not clear collection: {e}")  
  
 doc\_chunks = []  
 metadatas = []  
 ids = []  
  
 for idx, doc in enumerate(documents):  
 text = doc.get('text', '')  
 if not text or len(text) < 100:  
 continue  
  
 chunks = self.\_chunk\_text(text, CHUNK\_SIZE, CHUNK\_OVERLAP)  
  
 for chunk\_idx, chunk in enumerate(chunks):  
 doc\_chunks.append(chunk)  
 metadata = {  
 'url': doc.get('url', 'unknown'),  
 'title': doc.get('title', 'Untitled'),  
 'publish\_date': doc.get('publish\_dt'),  
 'chunk\_idx': chunk\_idx,  
 'source\_type': doc.get('mime', 'text/html')  
 }  
 metadatas.append(self.\_clean\_metadata(metadata))  
 ids.append(f"doc\_{idx}\_chunk\_{chunk\_idx}")  
  
 if doc\_chunks:  
 try:  
 self.collection.add(  
 documents=doc\_chunks,  
 metadatas=metadatas,  
 ids=ids  
 )  
 logger.info(f"✅ Indexed {len(doc\_chunks)} document chunks")  
 except Exception as e:  
 logger.error(f"Error adding to ChromaDB: {e}")  
 raise  
 else:  
 logger.warning("No chunks to index")  
  
 def \_chunk\_text(self, text: str, chunk\_size: int, overlap: int) -> List[str]:  
 chunks = []  
 start = 0  
  
 while start < len(text):  
 end = start + chunk\_size  
 chunk = text[start:end]  
  
 if end < len(text):  
 last\_period = chunk.rfind('.')  
 last\_newline = chunk.rfind('\n')  
 break\_point = max(last\_period, last\_newline)  
  
 if break\_point > chunk\_size // 2:  
 chunk = chunk[:break\_point + 1]  
 end = start + break\_point + 1  
  
 chunks.append(chunk.strip())  
 start = end - overlap  
  
 return [c for c in chunks if len(c) > 50]  
  
 def retrieve\_from\_graph(self, query\_type: str, filters: Dict) -> List[Dict]:  
 logger.info(f"Retrieving from graph: {query\_type}")  
 results = []  
  
 if query\_type == 'list\_facilities':  
 results = self.graph\_builder.query\_facilities(  
 division=filters.get('division'),  
 state=filters.get('state'),  
 status=filters.get('status')  
 )  
  
 elif query\_type == 'new\_expansions':  
 date\_from = filters.get('date\_from')  
 date\_to = filters.get('date\_to')  
  
 if date\_from and date\_to:  
 results = self.graph\_builder.query\_expansions(  
 date\_from=date\_from,  
 date\_to=date\_to  
 )  
 else:  
 # FIXED: Removed months\_back param to match graph\_builder signature  
 date\_to = datetime.now().date().isoformat()  
 date\_from = (datetime.now() - timedelta(days=730)).date().isoformat()  
 results = self.graph\_builder.query\_expansions(  
 date\_from=date\_from,  
 date\_to=date\_to  
 )  
  
 elif query\_type == 'hiring\_positions':  
 results = self.graph\_builder.query\_jobs(factory\_only=True)  
  
 logger.info(f"Retrieved {len(results)} results from graph")  
 return results  
  
 def retrieve\_from\_vector(self, query: str, n\_results: int = 5) -> List[Dict]:  
 try:  
 if self.collection.count() == 0:  
 logger.warning("Vector store is empty")  
 return []  
  
 results = self.collection.query(  
 query\_texts=[query],  
 n\_results=min(n\_results, self.collection.count())  
 )  
  
 passages = []  
 if results['documents'] and results['documents'][0]:  
 for doc, metadata, distance in zip(  
 results['documents'][0],  
 results['metadatas'][0],  
 results['distances'][0]  
 ):  
 passages.append({  
 'text': doc,  
 'metadata': metadata,  
 'relevance': 1 - distance  
 })  
  
 logger.info(f"Retrieved {len(passages)} passages from vector store")  
 return passages  
  
 except Exception as e:  
 logger.error(f"Vector retrieval error: {e}")  
 return []  
  
 def corroborate\_evidence(self, results: List[Dict]) -> List[Dict]:  
 if not results:  
 return results  
  
 if results and 'title' in results[0]:  
 return results # Job postings don’t need corroboration  
  
 facility\_groups = {}  
 for result in results:  
 facility\_name = (result.get('facility') or '').lower()  
 if not facility\_name:  
 continue  
 facility\_groups.setdefault(facility\_name, []).append(result)  
  
 corroborated\_results = []  
 for facility\_name, group in facility\_groups.items():  
 unique\_sources = set(r.get('url', '') for r in group if r.get('url'))  
 num\_sources = len(unique\_sources)  
  
 for result in group:  
 if num\_sources >= 2:  
 result['confidence'] = min(0.95, result.get('confidence', 0.5) + 0.2)  
 result['corroboration'] = f"Confirmed by {num\_sources} sources"  
 else:  
 result['corroboration'] = "Single source"  
 corroborated\_results.append(result)  
  
 return corroborated\_results

## src\rag\\_\_init.\_\_.py

# ==================================================  
# File: src/rag/\_\_init\_\_.py  
# ==================================================  
"""  
Retrieval-Augmented Generation module  
"""  
  
from .retriever import VectorRetriever  
from .query\_classifier import QueryClassifier  
from .generator import AnswerGenerator  
  
\_\_all\_\_ = ['VectorRetriever', 'QueryClassifier', 'AnswerGenerator']

## src\scrapers\address\_scraper.py

"""  
Address Directory Scraper - FIXED VERSION  
Scrapes https://www.motherson.com/contact/address-directory?country=India&page=X  
Critical for Query 1 - List Facilities  
"""  
  
import logging  
import json  
import time  
from pathlib import Path  
import requests  
from bs4 import BeautifulSoup  
from typing import List, Dict, Optional  
import re  
  
from src.config import CACHE\_DIR  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class AddressScraper:  
 """Scraper for Motherson address directory with proper table parsing"""  
   
 def \_\_init\_\_(self, rate\_limit: float = 2.0):  
 self.rate\_limit = rate\_limit  
 self.cache\_file = CACHE\_DIR / "address\_facilities.json"  
 self.base\_url = (  
 "https://www.motherson.com/contact/address-directory?country=India&page="  
 )  
 self.headers = {  
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36'  
 }  
   
 # Division mapping  
 self.division\_map = {  
 "MSWIL": "Wiring Systems",  
 "MSW": "Wiring Systems",  
 "WIRING": "Wiring Systems",  
 "SMP": "Polymers",  
 "SMR": "Vision Systems",  
 "VISION": "Vision Systems",  
 "MSSL": "Wiring Systems",  
 "PKC": "Wiring Systems",  
 "SEATING": "Seating Systems",  
 "LOGISTICS": "Logistics"  
 }  
   
 # City to state mapping  
 self.city\_to\_state = {  
 'Sanand': 'Gujarat', 'Ahmedabad': 'Gujarat', 'Navagam': 'Gujarat',  
 'Pune': 'Maharashtra', 'Chakan': 'Maharashtra', 'Mumbai': 'Maharashtra',  
 'Aurangabad': 'Maharashtra',  
 'Chennai': 'Tamil Nadu', 'Hosur': 'Tamil Nadu', 'Coimbatore': 'Tamil Nadu',  
 'Bangalore': 'Karnataka', 'Bengaluru': 'Karnataka',  
 'Manesar': 'Haryana', 'Gurgaon': 'Haryana', 'Gurugram': 'Haryana',  
 'Bawal': 'Haryana', 'Dharuhera': 'Haryana',  
 'Noida': 'Uttar Pradesh', 'Greater Noida': 'Uttar Pradesh',  
 'Haridwar': 'Uttarakhand',  
 'Hyderabad': 'Telangana',  
 'Kolkata': 'West Bengal'  
 }  
   
 def scrape\_all(self, pages: int = 19, use\_cache: bool = True) -> List[Dict]:  
 """Scrape all address-directory pages"""  
 if use\_cache and self.cache\_file.exists():  
 logger.info("📦 Using cached address directory results")  
 return json.loads(self.cache\_file.read\_text(encoding="utf-8"))  
   
 all\_facilities = []  
   
 for page in range(1, pages + 1):  
 url = f"{self.base\_url}{page}"  
 logger.info(f"🌐 Scraping page {page}/{pages}")  
   
 try:  
 resp = requests.get(url, headers=self.headers, timeout=30)  
 resp.raise\_for\_status()  
 facilities = self.\_parse\_page(resp.text)  
   
 if facilities:  
 all\_facilities.extend(facilities)  
 logger.info(f" ✅ Extracted {len(facilities)} facilities")  
 else:  
 logger.warning(f" ⚠️ No facilities found on page {page}")  
   
 except Exception as e:  
 logger.error(f" ✗ Failed page {page}: {e}")  
   
 time.sleep(self.rate\_limit)  
   
 # Save cache  
 self.cache\_file.write\_text(  
 json.dumps(all\_facilities, indent=2, ensure\_ascii=False),  
 encoding="utf-8"  
 )  
   
 logger.info(f"✅ Total facilities scraped: {len(all\_facilities)}")  
 return all\_facilities  
   
 def \_parse\_page(self, html: str) -> List[Dict]:  
 """  
 Parse facilities from page - FIXED VERSION  
 Handles multiple table structures and div-based layouts  
 """  
 soup = BeautifulSoup(html, "html.parser")  
 results = []  
   
 # Method 1: Try to find table with class or id  
 table = soup.find("table", class\_=re.compile(r'address|facility|location', re.I))  
 if not table:  
 table = soup.find("table")  
   
 if table:  
 logger.info(" 📋 Found table structure")  
 results = self.\_parse\_table(table)  
 if results:  
 return results  
   
 # Method 2: Try div-based card layout  
 logger.info(" 🔍 Trying card-based layout")  
 cards = soup.find\_all("div", class\_=re.compile(r'card|facility|location|address', re.I))  
   
 for card in cards:  
 facility = self.\_parse\_card(card)  
 if facility:  
 results.append(facility)  
   
 if results:  
 return results  
   
 # Method 3: Try list-based layout  
 logger.info(" 🔍 Trying list-based layout")  
 list\_items = soup.find\_all("li", class\_=re.compile(r'facility|location|address', re.I))  
   
 for item in list\_items:  
 facility = self.\_parse\_list\_item(item)  
 if facility:  
 results.append(facility)  
   
 return results  
   
 def \_parse\_table(self, table) -> List[Dict]:  
 """Parse table structure"""  
 results = []  
 rows = table.find\_all("tr")  
   
 if len(rows) < 2:  
 return results  
   
 # Try to identify header  
 header = rows[0]  
 header\_cells = [cell.get\_text(strip=True).lower() for cell in header.find\_all(["th", "td"])]  
   
 # Find column indices  
 name\_col = self.\_find\_col\_index(header\_cells, ['name', 'facility', 'company', 'location'])  
 city\_col = self.\_find\_col\_index(header\_cells, ['city', 'location', 'place'])  
 state\_col = self.\_find\_col\_index(header\_cells, ['state', 'region'])  
 address\_col = self.\_find\_col\_index(header\_cells, ['address', 'street'])  
   
 # Parse data rows  
 for row in rows[1:]:  
 cells = row.find\_all(["td", "th"])  
 if len(cells) < 2:  
 continue  
   
 cell\_texts = [cell.get\_text(strip=True) for cell in cells]  
   
 # Extract name  
 name = None  
 if name\_col is not None and name\_col < len(cell\_texts):  
 name = cell\_texts[name\_col]  
 elif len(cell\_texts) > 0:  
 name = cell\_texts[0]  
   
 if not name or len(name) < 3:  
 continue  
   
 # Extract location  
 city = None  
 state = None  
   
 if city\_col is not None and city\_col < len(cell\_texts):  
 location\_text = cell\_texts[city\_col]  
 city, state = self.\_split\_location(location\_text)  
   
 if state\_col is not None and state\_col < len(cell\_texts):  
 state = cell\_texts[state\_col]  
   
 # Extract address  
 address = None  
 if address\_col is not None and address\_col < len(cell\_texts):  
 address = cell\_texts[address\_col]  
   
 # Infer state from city  
 if city and not state:  
 state = self.city\_to\_state.get(city)  
   
 # Map division  
 division = self.\_map\_division(name)  
   
 results.append({  
 "name": name,  
 "division": division,  
 "city": city,  
 "state": state,  
 "address": address,  
 "country": "India",  
 "status": "operational",  
 "source": "address\_directory",  
 "date": None  
 })  
   
 return results  
   
 def \_parse\_card(self, card) -> Optional[Dict]:  
 """Parse card/div structure"""  
 text = card.get\_text(separator='|', strip=True)  
   
 # Try to find name in heading  
 name\_elem = card.find(['h3', 'h4', 'h5', 'strong', 'b'])  
 name = name\_elem.get\_text(strip=True) if name\_elem else None  
   
 if not name:  
 # Use first line  
 lines = text.split('|')  
 name = lines[0] if lines else None  
   
 if not name or len(name) < 3:  
 return None  
   
 # Extract location  
 city = self.\_extract\_city(text)  
 state = self.\_extract\_state(text)  
   
 if city and not state:  
 state = self.city\_to\_state.get(city)  
   
 # Map division  
 division = self.\_map\_division(text)  
   
 return {  
 "name": name,  
 "division": division,  
 "city": city,  
 "state": state,  
 "address": None,  
 "country": "India",  
 "status": "operational",  
 "source": "address\_directory",  
 "date": None  
 }  
   
 def \_parse\_list\_item(self, item) -> Optional[Dict]:  
 """Parse list item structure"""  
 return self.\_parse\_card(item) # Same logic  
   
 def \_find\_col\_index(self, headers: List[str], keywords: List[str]) -> Optional[int]:  
 """Find column index matching keywords"""  
 for idx, header in enumerate(headers):  
 if any(kw in header for kw in keywords):  
 return idx  
 return None  
   
 def \_split\_location(self, text: str) -> tuple:  
 """Split location text into city and state"""  
 if not text:  
 return None, None  
   
 parts = [p.strip() for p in text.split(",")]  
   
 if len(parts) == 2:  
 return parts[0], parts[1]  
 elif len(parts) > 2:  
 return parts[0], parts[-1]  
 else:  
 # Try to identify if it's a city or state  
 text\_clean = text.strip()  
 if text\_clean in self.city\_to\_state:  
 return text\_clean, self.city\_to\_state[text\_clean]  
 elif text\_clean in self.city\_to\_state.values():  
 return None, text\_clean  
 else:  
 return text\_clean, None  
   
 def \_extract\_city(self, text: str) -> Optional[str]:  
 """Extract city from text"""  
 for city in self.city\_to\_state.keys():  
 if city.lower() in text.lower():  
 return city  
 return None  
   
 def \_extract\_state(self, text: str) -> Optional[str]:  
 """Extract state from text"""  
 for state in self.city\_to\_state.values():  
 if state.lower() in text.lower():  
 return state  
 return None  
   
 def \_map\_division(self, text: str) -> str:  
 """Map division from text"""  
 text\_upper = text.upper()  
   
 for abbr, full\_name in self.division\_map.items():  
 if abbr in text\_upper:  
 return full\_name  
   
 # Keyword matching  
 if 'WIRING' in text\_upper or 'HARNESS' in text\_upper:  
 return 'Wiring Systems'  
 elif 'VISION' in text\_upper or 'MIRROR' in text\_upper:  
 return 'Vision Systems'  
 elif 'POLYMER' in text\_upper:  
 return 'Polymers'  
 elif 'SEATING' in text\_upper:  
 return 'Seating Systems'  
 elif 'LOGISTIC' in text\_upper:  
 return 'Logistics'  
   
 return 'Unknown'

## src\scrapers\base\_scraper.py

"""Base web scraper with rate limiting and caching"""  
import os  
import time  
import hashlib  
import json  
import requests  
import logging  
from datetime import datetime  
from typing import Dict, Optional, List  
from urllib.parse import urljoin, urlparse  
from urllib.robotparser import RobotFileParser  
from pathlib import Path  
  
from bs4 import BeautifulSoup  
import trafilatura  
  
from ..config import CACHE\_DIR, SCRAPER\_RATE\_LIMIT, SCRAPER\_TIMEOUT, SCRAPER\_MAX\_RETRIES  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class BaseScraper:  
 def \_\_init\_\_(self, cache\_dir: Path = None, rate\_limit: float = None):  
 self.cache\_dir = cache\_dir or CACHE\_DIR  
 self.rate\_limit = rate\_limit or SCRAPER\_RATE\_LIMIT  
 self.timeout = SCRAPER\_TIMEOUT  
 self.max\_retries = SCRAPER\_MAX\_RETRIES  
   
 self.last\_request\_time = {}  
 self.robot\_parsers = {}  
   
 self.headers = {  
 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36',  
 'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8',  
 }  
   
 def \_get\_cache\_path(self, url: str) -> Path:  
 url\_hash = hashlib.md5(url.encode()).hexdigest()  
 return self.cache\_dir / f"{url\_hash}.json"  
   
 def \_load\_from\_cache(self, url: str) -> Optional[Dict]:  
 cache\_path = self.\_get\_cache\_path(url)  
 if cache\_path.exists():  
 try:  
 with open(cache\_path, 'r', encoding='utf-8') as f:  
 return json.load(f)  
 except:  
 pass  
 return None  
   
 def \_save\_to\_cache(self, url: str, data: Dict):  
 try:  
 with open(self.\_get\_cache\_path(url), 'w', encoding='utf-8') as f:  
 json.dump(data, f, indent=2, ensure\_ascii=False, default=str)  
 except Exception as e:  
 logger.warning(f"Cache write error: {e}")  
   
 def \_check\_robots\_txt(self, url: str) -> bool:  
 parsed = urlparse(url)  
 domain = f"{parsed.scheme}://{parsed.netloc}"  
   
 if domain not in self.robot\_parsers:  
 robots\_url = urljoin(domain, '/robots.txt')  
 rp = RobotFileParser()  
 rp.set\_url(robots\_url)  
 try:  
 rp.read()  
 self.robot\_parsers[domain] = rp  
 except:  
 self.robot\_parsers[domain] = None  
   
 rp = self.robot\_parsers[domain]  
 return rp.can\_fetch("\*", url) if rp else True  
   
 def \_apply\_rate\_limit(self, domain: str):  
 if domain in self.last\_request\_time:  
 elapsed = time.time() - self.last\_request\_time[domain]  
 if elapsed < self.rate\_limit:  
 time.sleep(self.rate\_limit - elapsed)  
 self.last\_request\_time[domain] = time.time()  
   
 def \_fetch\_url(self, url: str) -> Optional[bytes]:  
 parsed = urlparse(url)  
 domain = parsed.netloc  
   
 if not self.\_check\_robots\_txt(url):  
 return None  
   
 self.\_apply\_rate\_limit(domain)  
   
 for attempt in range(self.max\_retries):  
 try:  
 response = requests.get(url, headers=self.headers, timeout=self.timeout)  
 response.raise\_for\_status()  
 return response.content  
 except Exception as e:  
 if attempt < self.max\_retries - 1:  
 time.sleep(2 \*\* attempt)  
 return None  
   
 def \_extract\_text\_from\_html(self, html\_content: bytes, url: str) -> Dict:  
 try:  
 text = trafilatura.extract(html\_content, include\_tables=True, no\_fallback=False)  
 soup = BeautifulSoup(html\_content, 'html.parser')  
 title = soup.title.string if soup.title else url  
   
 publish\_date = None  
 for tag in soup.find\_all(['time', 'meta']):  
 if tag.name == 'time' and tag.get('datetime'):  
 publish\_date = tag['datetime']  
 break  
   
 if not text:  
 for script in soup(['script', 'style', 'nav', 'footer']):  
 script.decompose()  
 text = soup.get\_text(separator='\n', strip=True)  
   
 return {  
 'url': url,  
 'text': text,  
 'title': title,  
 'fetched\_at': datetime.now().isoformat(),  
 'mime': 'text/html',  
 'publish\_dt': publish\_date  
 }  
 except Exception as e:  
 logger.error(f"HTML extraction error: {e}")  
 return None  
   
 def scrape\_url(self, url: str, use\_cache: bool = True) -> Optional[Dict]:  
 if use\_cache:  
 cached = self.\_load\_from\_cache(url)  
 if cached:  
 logger.info(f"Loaded from cache: {url}")  
 return cached  
   
 content = self.\_fetch\_url(url)  
 if not content:  
 return None  
   
 if url.lower().endswith('.pdf') or content[:4] == b'%PDF':  
 from .pdf\_extractor import PDFExtractor  
 extractor = PDFExtractor()  
 data = extractor.extract\_from\_bytes(content, url)  
 else:  
 data = self.\_extract\_text\_from\_html(content, url)  
   
 if data and use\_cache:  
 self.\_save\_to\_cache(url, data)  
   
 return data  
   
 def scrape\_multiple(self, urls: List[str], use\_cache: bool = True) -> List[Dict]:  
 results = []  
 for idx, url in enumerate(urls, 1):  
 logger.info(f"Scraping {idx}/{len(urls)}: {url}")  
 data = self.scrape\_url(url, use\_cache=use\_cache)  
 if data:  
 results.append(data)  
 return results

## src\scrapers\pdf\_extractor.py

"""  
Enhanced PDF Extractor - CRITICAL FIX FOR QUERY 2  
Now extracts expansion/greenfield data from MSWIL Annual Report  
"""  
import logging  
import re  
from datetime import datetime  
from typing import Dict, Optional, List  
from io import BytesIO  
import pdfplumber  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class PDFExtractor:  
 def \_\_init\_\_(self):  
 # Division mapping  
 self.division\_map = {  
 'MSWIL': 'Wiring Systems',  
 'MSW': 'Wiring Systems',  
 'WIRING': 'Wiring Systems',  
 'HARNESS': 'Wiring Systems',  
 'SMR': 'Vision Systems',  
 'SAMVARDHANA': 'Vision Systems',  
 'VISION': 'Vision Systems',  
 'MIRRORS': 'Vision Systems',  
 'SMP': 'Polymers',  
 'POLYMER': 'Polymers',  
 'PKC': 'Wiring Systems',  
 'SEATING': 'Seating Systems',  
 'LOGISTICS': 'Logistics'  
 }  
   
 # Indian cities to states mapping  
 self.city\_to\_state = {  
 'Sanand': 'Gujarat', 'Ahmedabad': 'Gujarat', 'Navagam': 'Gujarat',  
 'Pune': 'Maharashtra', 'Chakan': 'Maharashtra', 'Mumbai': 'Maharashtra',  
 'Chennai': 'Tamil Nadu', 'Hosur': 'Tamil Nadu',  
 'Bangalore': 'Karnataka', 'Bengaluru': 'Karnataka',  
 'Manesar': 'Haryana', 'Gurgaon': 'Haryana', 'Gurugram': 'Haryana',  
 'Noida': 'Uttar Pradesh', 'Haridwar': 'Uttarakhand',  
 'Bawal': 'Haryana', 'Dharuhera': 'Haryana',  
 'Hyderabad': 'Telangana', 'Kolkata': 'West Bengal'  
 }  
   
 # CRITICAL: Expansion keywords for Query 2  
 self.expansion\_keywords = [  
 'new plant', 'new facility', 'greenfield', 'brownfield',  
 'expansion', 'phase', 'establishing', 'set up', 'setting up',  
 'commenced operations', 'expected to commence', 'operational',  
 'inaugurated', 'announced', 'upcoming', 'proposed'  
 ]  
   
 def extract\_from\_bytes(self, pdf\_content: bytes, url: str) -> Optional[Dict]:  
 """Extract comprehensive data from PDF with EXPANSION FOCUS"""  
 try:  
 full\_text = []  
 facilities = []  
 expansions = [] # NEW: Track expansion-specific data  
   
 with pdfplumber.open(BytesIO(pdf\_content)) as pdf:  
 # Extract metadata  
 metadata = pdf.metadata or {}  
 title = metadata.get('Title', url)  
   
 for page\_num, page in enumerate(pdf.pages, 1):  
 # Extract text  
 page\_text = page.extract\_text()  
 if page\_text:  
 full\_text.append(f"[Page {page\_num}]\n{page\_text}")  
   
 # CRITICAL: Extract expansion mentions from text  
 page\_expansions = self.\_extract\_expansions\_from\_text(page\_text, page\_num)  
 expansions.extend(page\_expansions)  
   
 # Extract tables  
 tables = page.extract\_tables()  
 if tables:  
 for table\_idx, table in enumerate(tables):  
 table\_facilities = self.\_parse\_facility\_table(table, page\_num)  
 facilities.extend(table\_facilities)  
   
 # Add table to text  
 table\_text = self.\_table\_to\_text(table)  
 full\_text.append(f"\n[Table {table\_idx+1} on Page {page\_num}]\n{table\_text}")  
   
 # Parse creation date  
 creation\_date = metadata.get('CreationDate')  
 publish\_date = None  
 if creation\_date:  
 try:  
 date\_str = creation\_date.replace('D:', '').split('+')[0][:8]  
 if len(date\_str) == 8:  
 publish\_date = f"{date\_str[0:4]}-{date\_str[4:6]}-{date\_str[6:8]}"  
 except:  
 pass  
   
 combined\_text = '\n\n'.join(full\_text)  
   
 # Extract inline facilities  
 inline\_facilities = self.\_extract\_inline\_facilities(combined\_text)  
 facilities.extend(inline\_facilities)  
   
 # CRITICAL: Merge expansion data with facilities  
 facilities = self.\_merge\_expansion\_data(facilities, expansions)  
   
 # Add structured data to text  
 if facilities:  
 combined\_text += "\n\n=== EXTRACTED FACILITIES ===\n"  
 for fac in facilities:  
 combined\_text += f"Facility: {fac['name']}\n"  
 combined\_text += f" Division: {fac.get('division', 'Unknown')}\n"  
 combined\_text += f" Location: {fac.get('city', 'N/A')}, {fac.get('state', 'N/A')}\n"  
 combined\_text += f" Status: {fac.get('status', 'operational')}\n"  
   
 # CRITICAL: Add expansion info  
 if fac.get('expansion\_type'):  
 combined\_text += f" Expansion Type: {fac['expansion\_type']}\n"  
 if fac.get('date'):  
 combined\_text += f" Date: {fac['date']}\n"  
 if fac.get('timeline'):  
 combined\_text += f" Timeline: {fac['timeline']}\n"  
   
 combined\_text += "\n"  
   
 return {  
 'url': url,  
 'text': combined\_text,  
 'title': title,  
 'fetched\_at': datetime.now().isoformat(),  
 'mime': 'application/pdf',  
 'publish\_dt': publish\_date,  
 'structured\_facilities': facilities  
 }  
   
 except Exception as e:  
 logger.error(f"PDF extraction error: {e}")  
 import traceback  
 traceback.print\_exc()  
 return None  
   
 def \_extract\_expansions\_from\_text(self, text: str, page\_num: int) -> List[Dict]:  
 """CRITICAL: Extract expansion/greenfield mentions from text"""  
 expansions = []  
   
 # Pattern 1: "establishing X new plants in Y"  
 pattern1 = r'establishing\s+(\w+)\s+new\s+plants?\s+in\s+(\w+(?:\s+\([^)]+\))?(?:\s+and\s+\w+(?:\s+\([^)]+\))?)?)'  
   
 # Pattern 2: "new plant in City"  
 pattern2 = r'new\s+(?:plant|facility|unit)\s+in\s+(\w+)'  
   
 # Pattern 3: "City plant... operations in FY"  
 pattern3 = r'(\w+)\s+plant[^.]\*(?:commence|start|begin)\s+operations\s+in\s+(FY\s\*\d{4}[-–]\d{2,4}|Q\d\s+\d{4}|\d{4})'  
   
 # Pattern 4: "greenfield/brownfield in City"  
 pattern4 = r'(greenfield|brownfield)[^.]\*in\s+(\w+)'  
   
 # Pattern 5: Specific MSWIL expansion text  
 pattern5 = r'two\s+new\s+plants\s+in\s+([^.]+)\s+equipped\s+to\s+manufacture'  
   
 all\_patterns = [pattern1, pattern2, pattern3, pattern4, pattern5]  
   
 for pattern in all\_patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 # Extract location and timeline  
 matched\_text = match.group(0)  
   
 # Try to find cities  
 cities = []  
 for city in self.city\_to\_state.keys():  
 if city.lower() in matched\_text.lower():  
 cities.append(city)  
   
 # If pattern1, extract from groups  
 if pattern == pattern1:  
 try:  
 count = match.group(1)  
 locations = match.group(2)  
   
 # Parse "Navagam (Gujarat) and Pune (Maharashtra)"  
 location\_parts = re.findall(r'(\w+)\s\*\(([^)]+)\)', locations)  
 if location\_parts:  
 for city, state in location\_parts:  
 cities.append(city)  
 except:  
 pass  
   
 # Extract timeline  
 timeline = self.\_extract\_timeline\_from\_context(text, match.start(), match.end())  
   
 # Create expansion entries  
 if cities:  
 for city in cities:  
 state = self.city\_to\_state.get(city)  
 expansions.append({  
 'city': city,  
 'state': state,  
 'expansion\_type': 'greenfield' if 'greenfield' in matched\_text.lower() or 'new plant' in matched\_text.lower() else 'brownfield',  
 'status': self.\_infer\_status\_from\_text(matched\_text),  
 'timeline': timeline,  
 'page': page\_num,  
 'context': matched\_text  
 })  
 logger.info(f" 🔍 Found expansion: {city}, {state} - {timeline}")  
   
 return expansions  
   
 def \_extract\_timeline\_from\_context(self, text: str, start: int, end: int) -> Optional[str]:  
 """Extract timeline from surrounding context"""  
 # Get context window  
 context\_start = max(0, start - 500)  
 context\_end = min(len(text), end + 500)  
 context = text[context\_start:context\_end]  
   
 # Timeline patterns  
 patterns = [  
 r'FY\s\*(\d{4})[-–](\d{2,4})',  
 r'Q(\d)\s+(\d{4})',  
 r'(expected to|will|planned to)\s+commence\s+operations\s+in\s+(FY\s\*\d{4}[-–]\d{2,4})',  
 r'operational\s+(?:by|in)\s+(FY\s\*\d{4}[-–]\d{2,4}|Q\d\s+\d{4}|\d{4})',  
 r'(\d{4})[-–](\d{2,4})'  
 ]  
   
 for pattern in patterns:  
 match = re.search(pattern, context, re.IGNORECASE)  
 if match:  
 return match.group(0)  
   
 return None  
   
 def \_infer\_status\_from\_text(self, text: str) -> str:  
 """Infer facility status from text"""  
 text\_lower = text.lower()  
   
 if any(kw in text\_lower for kw in ['expected to commence', 'will commence', 'planned', 'upcoming', 'proposed']):  
 return 'planned'  
 elif any(kw in text\_lower for kw in ['under construction', 'being established', 'setting up']):  
 return 'under-construction'  
 elif any(kw in text\_lower for kw in ['operational', 'commenced', 'inaugurated']):  
 return 'operational'  
   
 return 'planned'  
   
 def \_merge\_expansion\_data(self, facilities: List[Dict], expansions: List[Dict]) -> List[Dict]:  
 """Merge expansion data into facilities"""  
 # Create map of cities to expansion data  
 expansion\_map = {}  
 for exp in expansions:  
 city = exp.get('city')  
 if city:  
 if city not in expansion\_map or exp.get('timeline'):  
 expansion\_map[city] = exp  
   
 # Merge into facilities  
 for fac in facilities:  
 city = fac.get('city')  
 if city and city in expansion\_map:  
 exp = expansion\_map[city]  
   
 # Add expansion info if not already present  
 if not fac.get('expansion\_type'):  
 fac['expansion\_type'] = exp.get('expansion\_type')  
 if not fac.get('timeline'):  
 fac['timeline'] = exp.get('timeline')  
   
 # Update status if expansion is more specific  
 if exp.get('status') and exp['status'] != 'operational':  
 fac['status'] = exp['status']  
   
 # Add standalone expansions that don't match existing facilities  
 for exp in expansions:  
 city = exp.get('city')  
   
 # Check if already in facilities  
 exists = any(f.get('city') == city for f in facilities)  
   
 if not exists:  
 # Create new facility entry from expansion  
 facilities.append({  
 'name': f"{city} Plant",  
 'division': 'Wiring Systems', # Default from MSWIL report  
 'city': city,  
 'state': exp.get('state'),  
 'status': exp.get('status', 'planned'),  
 'expansion\_type': exp.get('expansion\_type'),  
 'timeline': exp.get('timeline'),  
 'date': self.\_parse\_date(exp.get('timeline', '')),  
 'source\_type': 'expansion',  
 'page': exp.get('page')  
 })  
   
 return facilities  
   
 def \_parse\_facility\_table(self, table: List[List], page\_num: int) -> List[Dict]:  
 """Parse facility data from table"""  
 facilities = []  
   
 if not table or len(table) < 2:  
 return facilities  
   
 # Try to identify header row  
 header = table[0]  
 header\_lower = [str(cell).lower() if cell else '' for cell in header]  
   
 # Find column indices  
 facility\_col = self.\_find\_column(header\_lower, ['facility', 'plant', 'location', 'unit', 'site'])  
 division\_col = self.\_find\_column(header\_lower, ['division', 'business', 'segment'])  
 city\_col = self.\_find\_column(header\_lower, ['city', 'location', 'place'])  
 state\_col = self.\_find\_column(header\_lower, ['state', 'region'])  
 status\_col = self.\_find\_column(header\_lower, ['status', 'stage', 'phase'])  
 date\_col = self.\_find\_column(header\_lower, ['date', 'year', 'commissioned', 'operational'])  
   
 # Parse data rows  
 for row\_idx, row in enumerate(table[1:], 1):  
 if not row or len(row) < 2:  
 continue  
   
 facility\_data = {}  
   
 # Extract facility name  
 facility\_name = None  
 if facility\_col is not None and facility\_col < len(row):  
 facility\_name = self.\_clean\_cell(row[facility\_col])  
   
 if not facility\_name or len(facility\_name) < 3:  
 continue  
   
 facility\_data['name'] = facility\_name  
   
 # Extract division  
 if division\_col is not None and division\_col < len(row):  
 division = self.\_clean\_cell(row[division\_col])  
 facility\_data['division'] = self.\_map\_division(division)  
 else:  
 facility\_data['division'] = self.\_infer\_division(facility\_name)  
   
 # Extract location  
 city = None  
 state = None  
   
 if city\_col is not None and city\_col < len(row):  
 city = self.\_clean\_cell(row[city\_col])  
   
 if state\_col is not None and state\_col < len(row):  
 state = self.\_clean\_cell(row[state\_col])  
   
 # Parse combined location  
 if city and ',' in city:  
 parts = [p.strip() for p in city.split(',')]  
 city = parts[0]  
 if len(parts) > 1 and not state:  
 state = parts[1]  
   
 # Infer state from city  
 if city and not state and city in self.city\_to\_state:  
 state = self.city\_to\_state[city]  
   
 facility\_data['city'] = city  
 facility\_data['state'] = state  
   
 # Extract status  
 if status\_col is not None and status\_col < len(row):  
 status = self.\_clean\_cell(row[status\_col])  
 facility\_data['status'] = self.\_normalize\_status(status)  
 else:  
 facility\_data['status'] = 'operational'  
   
 # Extract date  
 if date\_col is not None and date\_col < len(row):  
 date\_text = self.\_clean\_cell(row[date\_col])  
 facility\_data['date'] = self.\_parse\_date(date\_text)  
   
 facility\_data['page'] = page\_num  
 facility\_data['source\_type'] = 'table'  
   
 facilities.append(facility\_data)  
   
 return facilities  
   
 def \_extract\_inline\_facilities(self, text: str) -> List[Dict]:  
 """Extract facilities from inline text"""  
 facilities = []  
   
 # Enhanced patterns  
 pattern1 = r'\b(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida|Gurgaon|Gurugram|Hyderabad|Mumbai)\s+(Plant|Facility|Unit|Manufacturing|Operations|Factory)\b'  
 pattern2 = r'\b(MSWIL|SMR|SMP|PKC)\s+(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida)\b'  
 pattern3 = r'\b(plant|facility|unit|manufacturing|operations)\s+(?:in|at|located in|located at)\s+(Sanand|Hosur|Chakan|Manesar|Pune|Ahmedabad|Chennai|Bangalore|Bengaluru|Navagam|Bawal|Haridwar|Noida|Gurgaon|Gurugram|Hyderabad|Mumbai)'  
   
 all\_patterns = [pattern1, pattern2, pattern3]  
   
 for pattern in all\_patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 matched\_text = match.group(0)  
   
 # Extract city  
 city = None  
 for city\_name in self.city\_to\_state.keys():  
 if city\_name.lower() in matched\_text.lower():  
 city = city\_name  
 break  
   
 if not city:  
 continue  
   
 # Build facility name  
 if 'MSWIL' in matched\_text.upper():  
 facility\_name = f"MSWIL {city} Plant"  
 elif 'SMR' in matched\_text.upper():  
 facility\_name = f"SMR {city} Plant"  
 elif 'SMP' in matched\_text.upper():  
 facility\_name = f"SMP {city} Plant"  
 else:  
 facility\_name = f"{city} Plant"  
   
 # Infer division  
 division = self.\_infer\_division(matched\_text)  
   
 # Get state  
 state = self.city\_to\_state.get(city)  
   
 # Find nearby date and status  
 context\_start = max(0, match.start() - 300)  
 context\_end = min(len(text), match.end() + 300)  
 context = text[context\_start:context\_end]  
   
 date = self.\_find\_date\_in\_context(context)  
 status = self.\_find\_status\_in\_context(context)  
   
 facilities.append({  
 'name': facility\_name,  
 'division': division,  
 'city': city,  
 'state': state,  
 'status': status or 'operational',  
 'date': date,  
 'source\_type': 'inline'  
 })  
   
 # Deduplicate  
 unique\_facilities = {}  
 for fac in facilities:  
 key = (fac['name'].lower(), fac.get('city', '').lower())  
 if key not in unique\_facilities:  
 unique\_facilities[key] = fac  
   
 return list(unique\_facilities.values())  
   
 def \_find\_column(self, header: List[str], keywords: List[str]) -> Optional[int]:  
 """Find column index matching keywords"""  
 for idx, cell in enumerate(header):  
 if any(kw in cell for kw in keywords):  
 return idx  
 return None  
   
 def \_clean\_cell(self, cell) -> str:  
 """Clean table cell"""  
 if cell is None:  
 return ''  
 text = str(cell).strip()  
 text = re.sub(r'\s+', ' ', text)  
 return text  
   
 def \_map\_division(self, division\_text: str) -> str:  
 """Map division abbreviations to full names"""  
 if not division\_text:  
 return 'Unknown'  
   
 division\_upper = division\_text.upper()  
   
 for abbr, full\_name in self.division\_map.items():  
 if abbr in division\_upper:  
 return full\_name  
   
 if 'WIRING' in division\_upper or 'HARNESS' in division\_upper:  
 return 'Wiring Systems'  
 elif 'VISION' in division\_upper or 'MIRROR' in division\_upper:  
 return 'Vision Systems'  
 elif 'POLYMER' in division\_upper:  
 return 'Polymers'  
 elif 'SEATING' in division\_upper:  
 return 'Seating Systems'  
 elif 'LOGISTIC' in division\_upper:  
 return 'Logistics'  
   
 return division\_text.title()  
   
 def \_infer\_division(self, text: str) -> str:  
 """Infer division from text"""  
 text\_upper = text.upper()  
   
 for abbr, full\_name in self.division\_map.items():  
 if abbr in text\_upper:  
 return full\_name  
   
 return 'Unknown'  
   
 def \_normalize\_status(self, status\_text: str) -> str:  
 """Normalize status"""  
 if not status\_text:  
 return 'operational'  
   
 status\_lower = status\_text.lower()  
   
 if any(kw in status\_lower for kw in ['plan', 'propos', 'upcom', 'futur', 'announc']):  
 return 'planned'  
 elif any(kw in status\_lower for kw in ['construction', 'building', 'develop']):  
 return 'under-construction'  
 else:  
 return 'operational'  
   
 def \_parse\_date(self, date\_text: str) -> Optional[str]:  
 """Parse date from text"""  
 if not date\_text:  
 return None  
   
 try:  
 # FY format  
 fy\_match = re.search(r'FY\s\*(\d{4})', date\_text, re.IGNORECASE)  
 if fy\_match:  
 year = fy\_match.group(1)  
 return f"{year}-03-31"  
   
 # Quarter format  
 q\_match = re.search(r'Q([1-4])\s+(\d{4})', date\_text, re.IGNORECASE)  
 if q\_match:  
 quarter = int(q\_match.group(1))  
 year = q\_match.group(2)  
 month = quarter \* 3  
 return f"{year}-{month:02d}-01"  
   
 # Year only  
 year\_match = re.search(r'\b(20\d{2})\b', date\_text)  
 if year\_match:  
 year = int(year\_match.group(1))  
 if 2010 <= year <= datetime.now().year + 5:  
 return f"{year}-01-01"  
   
 return None  
   
 except Exception as e:  
 logger.warning(f"Date parsing failed for '{date\_text}': {e}")  
 return None  
   
 def \_find\_date\_in\_context(self, context: str) -> Optional[str]:  
 """Find date in surrounding context"""  
 date\_patterns = [  
 r'FY\s\*(\d{4})',  
 r'Q([1-4])\s+(\d{4})',  
 r'\b(20\d{2})\b'  
 ]  
   
 for pattern in date\_patterns:  
 match = re.search(pattern, context, re.IGNORECASE)  
 if match:  
 return self.\_parse\_date(match.group(0))  
   
 return None  
   
 def \_find\_status\_in\_context(self, context: str) -> Optional[str]:  
 """Find status in surrounding context"""  
 context\_lower = context.lower()  
   
 if any(kw in context\_lower for kw in ['planned', 'proposed', 'upcoming', 'future', 'announced', 'expected to commence']):  
 return 'planned'  
 elif any(kw in context\_lower for kw in ['construction', 'building', 'under development', 'establishing']):  
 return 'under-construction'  
 elif any(kw in context\_lower for kw in ['operational', 'operating', 'commissioned', 'inaugurated']):  
 return 'operational'  
   
 return None  
   
 def \_table\_to\_text(self, table: List[List]) -> str:  
 """Convert table to readable text"""  
 lines = []  
 for row in table:  
 line = ' | '.join([str(cell) if cell else '' for cell in row])  
 lines.append(line)  
 return '\n'.join(lines)  
   
 def extract\_from\_file(self, filepath: str) -> Optional[Dict]:  
 """Extract from file path"""  
 try:  
 with open(filepath, 'rb') as f:  
 return self.extract\_from\_bytes(f.read(), filepath)  
 except Exception as e:  
 logger.error(f"File read error: {e}")  
 return None

## src\scrapers\selenium\_scraper.py

"""  
Selenium Scraper - ROBUST VERSION  
Better DOM selectors + regex fallback for factory job extraction  
"""  
  
import logging  
import re  
import time  
from datetime import datetime  
from typing import List, Dict, Optional  
  
from selenium import webdriver  
from selenium.webdriver.chrome.options import Options  
from selenium.webdriver.chrome.service import Service  
from selenium.webdriver.common.by import By  
from selenium.webdriver.support import expected\_conditions as EC  
from selenium.webdriver.support.ui import WebDriverWait  
from webdriver\_manager.chrome import ChromeDriverManager  
  
from src.config import FACTORY\_CAREER\_URL  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
  
class SeleniumScraper:  
 """Production-grade Selenium scraper with multiple strategies"""  
   
 def \_\_init\_\_(self, headless: bool = True):  
 self.headless = headless  
 self.factory\_keywords = [  
 'plant', 'production', 'manufacturing', 'assembly', 'operator',  
 'technician', 'mechanic', 'maintenance', 'quality', 'supervisor',  
 'foreman', 'shift', 'machine', 'tool', 'welder', 'fitter',  
 'warehouse', 'logistics', 'die', 'mold', 'press', 'injection',  
 'stamping', 'painting', 'welding', 'material', 'inventory',  
 'floor', 'line', 'process', 'supply chain'  
 ]  
   
 self.indian\_cities = [  
 'Chennai', 'Pune', 'Bangalore', 'Bengaluru', 'Hosur', 'Sanand',  
 'Manesar', 'Ahmedabad', 'Mumbai', 'Hyderabad', 'Chakan', 'Navagam',  
 'Gurgaon', 'Gurugram', 'Noida', 'Haridwar', 'Bawal', 'Dharuhera',  
 'Greater Noida', 'Aurangabad', 'Coimbatore', 'India'  
 ]  
   
 def \_init\_driver(self):  
 """Initialize ChromeDriver with robust settings"""  
 chrome\_options = Options()  
 if self.headless:  
 chrome\_options.add\_argument("--headless=new")  
 chrome\_options.add\_argument("--disable-gpu")  
 chrome\_options.add\_argument("--no-sandbox")  
 chrome\_options.add\_argument("--disable-dev-shm-usage")  
 chrome\_options.add\_argument("--window-size=1920,1080")  
 chrome\_options.add\_argument("--disable-blink-features=AutomationControlled")  
 chrome\_options.add\_argument("user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36")  
   
 try:  
 # FIXED: Remove os\_type parameter  
 service = Service(ChromeDriverManager().install())  
 driver = webdriver.Chrome(service=service, options=chrome\_options)  
 return driver  
 except Exception as e:  
 logger.error(f"ChromeDriver initialization failed: {e}")  
 # Fallback: try without service  
 driver = webdriver.Chrome(options=chrome\_options)  
 return driver  
   
 def scrape\_jobs(self, url: str = FACTORY\_CAREER\_URL) -> List[Dict]:  
 """Scrape jobs with multiple extraction strategies"""  
 logger.info(f"🔍 Scraping jobs from: {url}")  
 driver = self.\_init\_driver()  
   
 try:  
 driver.get(url)  
 time.sleep(3) # Wait for JavaScript to load  
   
 # Try multiple extraction strategies  
 jobs = []  
   
 # Strategy 1: Try structured DOM extraction (multiple selectors)  
 jobs = self.\_extract\_with\_dom\_strategy1(driver)  
 if not jobs:  
 jobs = self.\_extract\_with\_dom\_strategy2(driver)  
 if not jobs:  
 jobs = self.\_extract\_with\_dom\_strategy3(driver)  
   
 # Strategy 2: Fallback to regex extraction  
 if not jobs:  
 logger.warning("⚠️ DOM extraction failed, using regex fallback")  
 page\_text = driver.find\_element(By.TAG\_NAME, "body").text  
 jobs = self.\_extract\_with\_regex(page\_text)  
   
 # Strategy 3: Last resort - parse HTML source  
 if not jobs:  
 logger.warning("⚠️ Regex failed, parsing HTML source")  
 html\_source = driver.page\_source  
 jobs = self.\_extract\_from\_html(html\_source)  
   
 logger.info(f"✅ Extracted {len(jobs)} jobs")  
 return jobs  
   
 except Exception as e:  
 logger.error(f"❌ Job scraping failed: {e}")  
 import traceback  
 traceback.print\_exc()  
 return []  
   
 finally:  
 driver.quit()  
   
 def \_extract\_with\_dom\_strategy1(self, driver) -> List[Dict]:  
 """Strategy 1: Common job board selectors"""  
 try:  
 wait = WebDriverWait(driver, 10)  
   
 # Try common selectors  
 selectors = [  
 "div.job-item",  
 "div.job-card",  
 "div.position-item",  
 "div.career-item",  
 "li.job-listing",  
 "tr.job-row",  
 "div[class\*='job']",  
 "div[data-job-id]"  
 ]  
   
 jobs = []  
 for selector in selectors:  
 try:  
 elements = driver.find\_elements(By.CSS\_SELECTOR, selector)  
 if elements:  
 logger.info(f"✅ Found {len(elements)} jobs with selector: {selector}")  
   
 for elem in elements:  
 try:  
 # Extract job data  
 title = self.\_extract\_text(elem, ["a.job-title", "h3", "h4", ".title", "a[href\*='job']"])  
 location = self.\_extract\_text(elem, [".location", ".job-location", "span.city", ".place"])  
 department = self.\_extract\_text(elem, [".department", ".category", ".division"])  
 link = self.\_extract\_link(elem, ["a.job-title", "a.job-link", "a[href\*='job']"])  
   
 if title and self.\_is\_factory\_job(title):  
 jobs.append({  
 'title': title.strip(),  
 'location': location or self.\_guess\_location(elem.text),  
 'department': department,  
 'url': link or FACTORY\_CAREER\_URL,  
 'source': 'motherson\_careers'  
 })  
 except:  
 continue  
   
 if jobs:  
 return jobs  
 except:  
 continue  
   
 return jobs  
   
 except Exception as e:  
 logger.warning(f"Strategy 1 failed: {e}")  
 return []  
   
 def \_extract\_with\_dom\_strategy2(self, driver) -> List[Dict]:  
 """Strategy 2: Table-based extraction"""  
 try:  
 tables = driver.find\_elements(By.TAG\_NAME, "table")  
   
 jobs = []  
 for table in tables:  
 rows = table.find\_elements(By.TAG\_NAME, "tr")  
   
 for row in rows[1:]: # Skip header  
 cells = row.find\_elements(By.TAG\_NAME, "td")  
   
 if len(cells) >= 2:  
 title = cells[0].text.strip()  
 location = cells[1].text.strip() if len(cells) > 1 else 'India'  
   
 if title and self.\_is\_factory\_job(title):  
 jobs.append({  
 'title': title,  
 'location': location,  
 'department': None,  
 'url': FACTORY\_CAREER\_URL,  
 'source': 'motherson\_careers\_table'  
 })  
   
 if jobs:  
 logger.info(f"✅ Strategy 2 found {len(jobs)} jobs")  
 return jobs  
   
 except Exception as e:  
 logger.warning(f"Strategy 2 failed: {e}")  
 return []  
   
 def \_extract\_with\_dom\_strategy3(self, driver) -> List[Dict]:  
 """Strategy 3: List-based extraction"""  
 try:  
 lists = driver.find\_elements(By.CSS\_SELECTOR, "ul, ol")  
   
 jobs = []  
 for list\_elem in lists:  
 items = list\_elem.find\_elements(By.TAG\_NAME, "li")  
   
 for item in items:  
 text = item.text.strip()  
   
 # Check if it looks like a job posting  
 if len(text) > 10 and len(text) < 200:  
 # Try to split title and location  
 parts = re.split(r'[-|–—]', text, maxsplit=1)  
 title = parts[0].strip()  
 location = parts[1].strip() if len(parts) > 1 else 'India'  
   
 if self.\_is\_factory\_job(title):  
 jobs.append({  
 'title': title,  
 'location': location or self.\_guess\_location(text),  
 'department': None,  
 'url': FACTORY\_CAREER\_URL,  
 'source': 'motherson\_careers\_list'  
 })  
   
 if jobs:  
 logger.info(f"✅ Strategy 3 found {len(jobs)} jobs")  
 return jobs  
   
 except Exception as e:  
 logger.warning(f"Strategy 3 failed: {e}")  
 return []  
   
 def \_extract\_with\_regex(self, text: str) -> List[Dict]:  
 """Regex-based extraction as fallback"""  
 jobs = []  
   
 # Enhanced job patterns  
 patterns = [  
 r'(Production|Manufacturing|Assembly|Plant|Quality|Maintenance|Warehouse|Logistics|Tool|Die|Mold|Process|Line)\s+(Manager|Engineer|Supervisor|Operator|Technician|Coordinator|Specialist|Planner|Designer|Controller|Lead|Head|In-charge)',  
 r'(Senior|Junior|Lead|Chief|Deputy|Assistant|Sr\.|Jr\.)\s+\w+\s+(Engineer|Manager|Supervisor|Coordinator|Technician|Specialist)',  
 r'\w+\s+(Operator|Technician|Mechanic|Fitter|Welder|Assembler|Inspector|Machinist)',  
 r'(Shift|Floor|Line|Production|Process|Material)\s+(Manager|Supervisor|Coordinator|In-charge|Lead|Engineer)',  
 r'(Inventory|Supply Chain|Stamping|Welding|Painting|Injection|Molding)\s+(Engineer|Manager|Supervisor|Technician|Specialist)'  
 ]  
   
 for pattern in patterns:  
 for match in re.finditer(pattern, text, re.IGNORECASE):  
 title = match.group(0).strip()  
   
 if self.\_is\_factory\_job(title):  
 # Try to find location nearby  
 context\_start = max(0, match.start() - 200)  
 context\_end = min(len(text), match.end() + 200)  
 context = text[context\_start:context\_end]  
   
 location = self.\_guess\_location(context)  
   
 jobs.append({  
 'title': title,  
 'location': location,  
 'department': None,  
 'url': FACTORY\_CAREER\_URL,  
 'source': 'regex\_extraction'  
 })  
   
 # Deduplicate  
 unique\_jobs = []  
 seen\_titles = set()  
 for job in jobs:  
 title\_lower = job['title'].lower()  
 if title\_lower not in seen\_titles:  
 seen\_titles.add(title\_lower)  
 unique\_jobs.append(job)  
   
 if unique\_jobs:  
 logger.info(f"✅ Regex extraction found {len(unique\_jobs)} jobs")  
 return unique\_jobs  
   
 def \_extract\_from\_html(self, html: str) -> List[Dict]:  
 """Parse HTML source as last resort"""  
 from bs4 import BeautifulSoup  
   
 jobs = []  
 soup = BeautifulSoup(html, 'html.parser')  
   
 # Look for job-related elements  
 job\_elements = soup.find\_all(['div', 'li', 'tr'], class\_=re.compile(r'job|position|career|vacancy', re.I))  
   
 for elem in job\_elements:  
 text = elem.get\_text(strip=True)  
   
 # Extract title  
 title\_elem = elem.find(['h3', 'h4', 'a', 'span'], class\_=re.compile(r'title|name', re.I))  
 title = title\_elem.get\_text(strip=True) if title\_elem else text[:100]  
   
 # Extract location  
 loc\_elem = elem.find(['span', 'div'], class\_=re.compile(r'location|city|place', re.I))  
 location = loc\_elem.get\_text(strip=True) if loc\_elem else self.\_guess\_location(text)  
   
 if title and self.\_is\_factory\_job(title):  
 jobs.append({  
 'title': title,  
 'location': location,  
 'department': None,  
 'url': FACTORY\_CAREER\_URL,  
 'source': 'html\_parsing'  
 })  
   
 if jobs:  
 logger.info(f"✅ HTML parsing found {len(jobs)} jobs")  
 return jobs  
   
 def \_extract\_text(self, element, selectors: List[str]) -> Optional[str]:  
 """Try multiple selectors to extract text"""  
 for selector in selectors:  
 try:  
 elem = element.find\_element(By.CSS\_SELECTOR, selector)  
 text = elem.text.strip()  
 if text:  
 return text  
 except:  
 continue  
   
 # Fallback: return element text  
 try:  
 return element.text.strip()  
 except:  
 return None  
   
 def \_extract\_link(self, element, selectors: List[str]) -> Optional[str]:  
 """Try multiple selectors to extract link"""  
 for selector in selectors:  
 try:  
 elem = element.find\_element(By.CSS\_SELECTOR, selector)  
 href = elem.get\_attribute('href')  
 if href:  
 return href  
 except:  
 continue  
 return None  
   
 def \_is\_factory\_job(self, title: str) -> bool:  
 """Check if job title is factory-related"""  
 if not title or len(title) < 5:  
 return False  
   
 title\_lower = title.lower()  
   
 # Exclude non-factory roles  
 non\_factory\_keywords = [  
 'software', 'developer', 'programmer', 'data scientist',  
 'it ', 'digital', 'cyber', 'application', 'web', 'mobile',  
 'cloud', 'devops', 'analyst', 'sap', 'erp', 'finance',  
 'hr', 'marketing', 'sales', 'legal', 'admin'  
 ]  
   
 if any(kw in title\_lower for kw in non\_factory\_keywords):  
 return False  
   
 # Check for factory keywords  
 return any(kw in title\_lower for kw in self.factory\_keywords)  
   
 def \_guess\_location(self, text: str) -> str:  
 """Guess location from text"""  
 if not text:  
 return 'India'  
   
 text\_lower = text.lower()  
   
 # Check for city names  
 for city in self.indian\_cities:  
 if city.lower() in text\_lower:  
 return city  
   
 return 'India'  
  
  
# Regex fallback class (for backward compatibility)  
class RegexFallbackScraper:  
 """Standalone regex-based scraper"""  
   
 def \_\_init\_\_(self):  
 self.factory\_keywords = [  
 'plant', 'production', 'manufacturing', 'assembly', 'operator',  
 'technician', 'mechanic', 'maintenance', 'quality', 'supervisor',  
 'foreman', 'shift', 'machine', 'tool', 'welder', 'fitter',  
 'warehouse', 'logistics', 'die', 'mold', 'press', 'injection',  
 'stamping', 'painting', 'welding', 'material', 'inventory',  
 'floor', 'line'  
 ]  
 self.indian\_cities = [  
 'Chennai', 'Pune', 'Bangalore', 'Bengaluru', 'Hosur', 'Sanand',  
 'Manesar', 'Ahmedabad', 'Mumbai', 'Hyderabad', 'Chakan', 'Navagam',  
 'Gurgaon', 'Gurugram', 'Noida', 'Haridwar', 'Bawal', 'Dharuhera',  
 'Greater Noida', 'Aurangabad', 'Coimbatore'  
 ]  
   
 def extract\_from\_text(self, text: str) -> List[Dict]:  
 """Use regex to identify likely factory job listings"""  
 jobs = []  
 job\_patterns = [  
 r'(Production|Manufacturing|Assembly|Plant|Quality|Maintenance|Warehouse|Logistics|Tool|Die|Mold)\s+(Manager|Engineer|Supervisor|Operator|Technician)',  
 r'(Shift|Line|Floor|Material)\s+(Manager|Supervisor|Lead|Coordinator)',  
 r'(Technician|Mechanic|Fitter|Welder|Assembler|Inspector|Machinist)'  
 ]  
   
 for pattern in job\_patterns:  
 matches = re.finditer(pattern, text, re.IGNORECASE)  
 for m in matches:  
 title = m.group(0).strip()  
 if self.\_is\_valid\_job(title):  
 jobs.append({"title": title, "location": self.\_guess\_location(text)})  
   
 # Deduplicate  
 unique = []  
 seen = set()  
 for job in jobs:  
 title = job["title"].lower()  
 if title not in seen:  
 seen.add(title)  
 unique.append(job)  
 return unique  
   
 def \_is\_valid\_job(self, title: str) -> bool:  
 title\_lower = title.lower()  
 if any(kw in title\_lower for kw in ['software', 'developer', 'it', 'digital']):  
 return False  
 return any(kw in title\_lower for kw in self.factory\_keywords)  
   
 def \_guess\_location(self, text: str) -> str:  
 for city in self.indian\_cities:  
 if city.lower() in text.lower():  
 return city  
 return "India"

## src\scrapers\\_\_init\_\_.py

# ==================================================  
# File: src/scrapers/\_\_init\_\_.py  
# ==================================================  
"""  
Web scraping and PDF extraction module  
"""  
  
from .base\_scraper import BaseScraper  
from .pdf\_extractor import PDFExtractor  
  
\_\_all\_\_ = ['BaseScraper', 'PDFExtractor']

## src\ui\app.py

"""  
Streamlit App - COMPLETE WORKING VERSION  
No errors, all imports included  
"""  
  
import streamlit as st  
import pandas as pd  
import logging  
from datetime import datetime, timedelta  
from typing import List, Dict  
import sys  
from pathlib import Path  
  
# Add parent directory to path  
sys.path.insert(0, str(Path(\_\_file\_\_).parent.parent.parent))  
  
from src.rag.retriever import Retriever  
from src.rag.query\_classifier import QueryClassifier  
from src.rag.generator import Generator  
  
# Setup logging  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(\_\_name\_\_)  
  
# Page config  
st.set\_page\_config(  
 page\_title="Motherson Intelligence Platform",  
 page\_icon="🏭",  
 layout="wide"  
)  
  
# Custom CSS  
st.markdown("""  
<style>  
.main-header {  
 font-size: 2.5rem;  
 font-weight: bold;  
 color: #1f2937;  
 margin-bottom: 0.5rem;  
}  
.sub-header {  
 font-size: 1.1rem;  
 color: #6b7280;  
 margin-bottom: 2rem;  
}  
.evidence-box {  
 background-color: #f9fafb;  
 border-left: 4px solid #3b82f6;  
 padding: 1rem;  
 margin-bottom: 1rem;  
 border-radius: 0.5rem;  
}  
.highlight {  
 background-color: #fef3c7;  
 padding: 0.125rem 0.25rem;  
 border-radius: 0.25rem;  
 font-weight: 600;  
}  
.citation {  
 background-color: #3b82f6;  
 color: white;  
 padding: 0.125rem 0.375rem;  
 border-radius: 0.25rem;  
 font-size: 0.75rem;  
 font-weight: bold;  
 text-decoration: none;  
 vertical-align: super;  
}  
.confidence-high {  
 background-color: #d1fae5;  
 color: #065f46;  
 padding: 0.25rem 0.75rem;  
 border-radius: 1rem;  
 font-size: 0.875rem;  
 font-weight: 600;  
}  
.confidence-medium {  
 background-color: #fef3c7;  
 color: #92400e;  
 padding: 0.25rem 0.75rem;  
 border-radius: 1rem;  
 font-size: 0.875rem;  
 font-weight: 600;  
}  
.confidence-low {  
 background-color: #fee2e2;  
 color: #991b1b;  
 padding: 0.25rem 0.75rem;  
 border-radius: 1rem;  
 font-size: 0.875rem;  
 font-weight: 600;  
}  
</style>  
""", unsafe\_allow\_html=True)  
  
# Initialize components  
@st.cache\_resource  
def init\_components():  
 retriever = Retriever()  
 classifier = QueryClassifier()  
 generator = Generator()  
 return retriever, classifier, generator  
  
retriever, classifier, generator = init\_components()  
  
# Session state  
if 'current\_results' not in st.session\_state:  
 st.session\_state.current\_results = None  
if 'current\_evidence' not in st.session\_state:  
 st.session\_state.current\_evidence = []  
if 'current\_query\_type' not in st.session\_state:  
 st.session\_state.current\_query\_type = None  
if 'current\_answer' not in st.session\_state:  
 st.session\_state.current\_answer = None  
if 'current\_warning' not in st.session\_state:  
 st.session\_state.current\_warning = None  
  
# Helper functions  
def get\_confidence\_badge(confidence: float) -> str:  
 """Return HTML badge for confidence score"""  
 if confidence >= 0.8:  
 return f'<span class="confidence-high">{confidence:.2f}</span>'  
 elif confidence >= 0.5:  
 return f'<span class="confidence-medium">{confidence:.2f}</span>'  
 else:  
 return f'<span class="confidence-low">{confidence:.2f}</span>'  
  
def format\_location(city: str, state: str) -> str:  
 """Format location string"""  
 city = city or ''  
 state = state or ''  
   
 if city and state:  
 return f"{city}, {state}"  
 elif city:  
 return city  
 elif state:  
 return state  
 return "N/A"  
  
def deduplicate\_results(results: List[Dict]) -> List[Dict]:  
 """  
 Remove duplicates - FIXED VERSION  
 More lenient deduplication to preserve more results  
 """  
 if not results:  
 return []  
   
 seen = set()  
 deduplicated = []  
   
 for result in results:  
 # Build unique key based on query type  
 if 'facility' in result or 'name' in result:  
 facility = (result.get('facility') or result.get('name') or '').strip()  
 city = (result.get('city') or '').strip()  
   
 # Only dedupe if BOTH facility AND city match  
 key = (facility.lower(), city.lower())  
   
 # Skip completely empty entries  
 if key == ('', ''):  
 continue  
   
 elif 'title' in result:  
 title = (result.get('title') or '').strip()  
 location = (result.get('location') or '').strip()  
   
 # Only dedupe if BOTH title AND location match  
 key = (title.lower(), location.lower())  
   
 # Skip completely empty entries  
 if key == ('', ''):  
 continue  
 else:  
 # For other result types, use string representation  
 key = str(result)  
   
 if key not in seen:  
 seen.add(key)  
 deduplicated.append(result)  
   
 return deduplicated  
  
def execute\_query(query\_text: str = None, query\_type: str = None, filters: Dict = None):  
 """  
 Execute query and update session state - COMPLETELY FIXED  
 """  
 with st.spinner("🔍 Searching database..."):  
 # Classify query if needed  
 if query\_text and not query\_type:  
 query\_type = classifier.classify(query\_text)  
   
 st.session\_state.current\_query\_type = query\_type  
   
 # DEBUG: Show what we're querying  
 logger.info(f"🔍 Executing query: type={query\_type}, filters={filters}")  
   
 # Retrieve from graph  
 graph\_results = retriever.retrieve\_from\_graph(query\_type, filters or {})  
   
 # DEBUG: Log raw results  
 logger.info(f"📊 Raw results from graph: {len(graph\_results)}")  
   
 # CRITICAL FIX: Check if we have results BEFORE deduplication  
 if not graph\_results or len(graph\_results) == 0:  
 st.session\_state.current\_results = []  
 st.session\_state.current\_evidence = []  
 st.session\_state.current\_answer = None  
 st.session\_state.current\_warning = f"⚠️ No data found for query type '{query\_type}'. Database has data but query returned nothing."  
 logger.warning(f"❌ No results from query\_type={query\_type}")  
 return  
   
 # Deduplicate with fixed logic  
 deduplicated = deduplicate\_results(graph\_results)  
   
 # DEBUG: Log after deduplication  
 logger.info(f"📊 After deduplication: {len(deduplicated)} results")  
   
 # CRITICAL FIX: Check again after deduplication with better message  
 if not deduplicated or len(deduplicated) == 0:  
 st.session\_state.current\_results = []  
 st.session\_state.current\_evidence = []  
 st.session\_state.current\_answer = None  
 st.session\_state.current\_warning = f"⚠️ Found {len(graph\_results)} results but all were filtered as duplicates. Try adjusting filters."  
 logger.warning(f"❌ All {len(graph\_results)} results filtered during deduplication")  
 return  
   
 # Corroborate evidence  
 corroborated = retriever.corroborate\_evidence(deduplicated)  
   
 # Apply guardrails  
 processed = generator.apply\_guardrails(corroborated, query\_type)  
   
 # DEBUG: Log final processed count  
 logger.info(f"📊 Final processed: {len(processed['data'])} results")  
   
 # Generate answer if LLM available  
 if query\_text and len(processed['data']) > 0:  
 try:  
 vector\_passages = retriever.retrieve\_from\_vector(query\_text)  
 answer = generator.generate\_answer(query\_text, processed['data'], vector\_passages)  
 processed['answer'] = answer  
 except Exception as e:  
 logger.error(f"LLM generation failed: {e}")  
 processed['answer'] = None  
   
 # Update session state  
 st.session\_state.current\_results = processed['data']  
 st.session\_state.current\_evidence = processed['evidence']  
 st.session\_state.current\_answer = processed.get('answer')  
 st.session\_state.current\_warning = processed.get('warning')  
   
 logger.info(f"✅ Query complete: {len(processed['data'])} results, {len(processed['evidence'])} evidence items")  
  
# Header  
st.markdown('<div class="main-header">🏭 Motherson Intelligence Platform</div>', unsafe\_allow\_html=True)  
st.markdown('<div class="sub-header">AI-Powered India Facility Intelligence with Evidence-Based Insights</div>', unsafe\_allow\_html=True)  
  
# Sidebar - Filters  
with st.sidebar:  
 st.header("🔍 Filters")  
 st.markdown("---")  
   
 # Division filter  
 division\_filter = st.selectbox(  
 "Division",  
 ["All", "Wiring Systems", "Vision Systems", "Seating Systems", "Polymers", "Logistics"],  
 key="division\_filter"  
 )  
   
 # State filter  
 state\_filter = st.selectbox(  
 "State (India)",  
 ["All", "Gujarat", "Tamil Nadu", "Maharashtra", "Haryana", "Karnataka",  
 "Uttar Pradesh", "Rajasthan", "Punjab", "Telangana", "Andhra Pradesh",  
 "West Bengal", "Madhya Pradesh", "Kerala", "Odisha", "Uttarakhand"],  
 key="state\_filter"  
 )  
   
 # Status filter  
 status\_filter = st.selectbox(  
 "Status",  
 ["All", "operational", "under-construction", "planned"],  
 key="status\_filter"  
 )  
   
 st.markdown("---")  
   
 # Date range filter  
 st.subheader("📅 Date Range")  
 col1, col2 = st.columns(2)  
 with col1:  
 date\_from = st.date\_input(  
 "From",  
 value=datetime.now() - timedelta(days=730),  
 key="date\_from"  
 )  
 with col2:  
 date\_to = st.date\_input(  
 "To",  
 value=datetime.now(),  
 key="date\_to"  
 )  
   
 st.markdown("---")  
   
 # Apply filters button  
 if st.button("✅ Apply Filters", use\_container\_width=True, type="primary"):  
 if st.session\_state.current\_query\_type:  
 filters = {  
 'division': None if division\_filter == "All" else division\_filter,  
 'state': None if state\_filter == "All" else state\_filter,  
 'status': None if status\_filter == "All" else status\_filter,  
 'date\_from': date\_from.isoformat() if date\_from else None,  
 'date\_to': date\_to.isoformat() if date\_to else None  
 }  
 execute\_query(query\_type=st.session\_state.current\_query\_type, filters=filters)  
 else:  
 st.warning("Please run a query first!")  
  
# Build filters dict  
filters = {  
 'division': None if division\_filter == "All" else division\_filter,  
 'state': None if state\_filter == "All" else state\_filter,  
 'status': None if status\_filter == "All" else status\_filter,  
 'date\_from': date\_from.isoformat() if date\_from else None,  
 'date\_to': date\_to.isoformat() if date\_to else None  
}  
  
# Main content - Search section  
st.subheader("🔎 Search Query")  
  
col1, col2 = st.columns([4, 1])  
  
with col1:  
 custom\_query = st.text\_input(  
 "Enter your query about Motherson India facilities",  
 placeholder="e.g., Show wiring facilities in Gujarat with expansion plans",  
 label\_visibility="collapsed",  
 key="custom\_query"  
 )  
  
with col2:  
 if st.button("🔍 Search", use\_container\_width=True, type="primary"):  
 if custom\_query:  
 execute\_query(query\_text=custom\_query, filters=filters)  
 else:  
 st.warning("Please enter a query!")  
  
# Preset query buttons  
st.subheader("📋 Preset Queries")  
col1, col2, col3 = st.columns(3)  
  
with col1:  
 if st.button(  
 "📍 Query 1: List All Facilities",  
 help="List Motherson India facilities by division with location, status, and dates",  
 use\_container\_width=True,  
 key="query1"  
 ):  
 execute\_query(  
 query\_text="List all Motherson India facilities by division with location and status",  
 query\_type="list\_facilities",  
 filters=filters  
 )  
  
with col2:  
 if st.button(  
 "🏗️ Query 2: New/Expanded Plants",  
 help="Show greenfield or expansion projects in the last 24 months with timeline",  
 use\_container\_width=True,  
 key="query2"  
 ):  
 execute\_query(  
 query\_text="Show new or expanded plants in India in the last 24 months",  
 query\_type="new\_expansions",  
 filters=filters  
 )  
  
with col3:  
 if st.button(  
 "👷 Query 3: Hiring Positions",  
 help="Surface factory hiring positions at new facilities (excludes IT/software roles)",  
 use\_container\_width=True,  
 key="query3"  
 ):  
 execute\_query(  
 query\_text="Show hiring positions for factory roles in India",  
 query\_type="hiring\_positions",  
 filters=filters  
 )  
  
# Display results  
if st.session\_state.current\_results is not None:  
 st.markdown("---")  
   
 # Show warning if any  
 if st.session\_state.current\_warning:  
 st.warning(st.session\_state.current\_warning)  
   
 # LLM Answer  
 if st.session\_state.current\_answer:  
 st.subheader("💡 AI-Generated Summary")  
 st.info(st.session\_state.current\_answer)  
 st.markdown("---")  
   
 # Results table  
 st.subheader("📊 Results")  
   
 # Show count even if 0  
 result\_count = len(st.session\_state.current\_results) if st.session\_state.current\_results else 0  
 logger.info(f"🖥️ Displaying {result\_count} results to user")  
   
 if result\_count > 0:  
 # Create DataFrame based on query type  
 if st.session\_state.current\_query\_type == "list\_facilities":  
 df\_data = []  
 for idx, result in enumerate(st.session\_state.current\_results):  
 df\_data.append({  
 'Division': result.get('division', 'N/A'),  
 'Facility': result.get('facility') or result.get('name', 'N/A'),  
 'Location': format\_location(result.get('city'), result.get('state')),  
 'Status': result.get('status', 'N/A'),  
 'First Date': result.get('first\_date') or result.get('last\_event\_date', 'N/A'),  
 'Confidence': result.get('confidence', 0.9),  
 'Citation': f'<a href="#ev-{idx}" class="citation">[{idx + 1}]</a>'  
 })  
   
 df = pd.DataFrame(df\_data)  
 df['Confidence'] = df['Confidence'].apply(  
 lambda x: get\_confidence\_badge(x) if isinstance(x, (int, float)) else x  
 )  
 st.markdown(df.to\_html(escape=False, index=False), unsafe\_allow\_html=True)  
   
 elif st.session\_state.current\_query\_type == "new\_expansions":  
 df\_data = []  
 for idx, result in enumerate(st.session\_state.current\_results):  
 df\_data.append({  
 'Facility': result.get('facility', 'N/A'),  
 'Division': result.get('division', 'N/A'),  
 'Type': result.get('expansion\_type', 'N/A'),  
 'Location': format\_location(result.get('city'), result.get('state')),  
 'Timeline': result.get('timeline') or result.get('event\_date', 'N/A'),  
 'Confidence': result.get('confidence', 0.8),  
 'Citation': f'<a href="#ev-{idx}" class="citation">[{idx + 1}]</a>'  
 })  
   
 df = pd.DataFrame(df\_data)  
 df['Confidence'] = df['Confidence'].apply(  
 lambda x: get\_confidence\_badge(x) if isinstance(x, (int, float)) else x  
 )  
 st.markdown(df.to\_html(escape=False, index=False), unsafe\_allow\_html=True)  
   
 elif st.session\_state.current\_query\_type == "hiring\_positions":  
 df\_data = []  
 for idx, result in enumerate(st.session\_state.current\_results):  
 df\_data.append({  
 'Job Title': result.get('title', 'N/A'),  
 'Location': result.get('location', 'N/A'),  
 'Facility': result.get('facility', 'N/A'),  
 'Division': result.get('division', 'N/A'),  
 'Factory Role': '✓' if result.get('is\_factory\_role') else '✗',  
 'Citation': f'<a href="#ev-{idx}" class="citation">[{idx + 1}]</a>'  
 })  
   
 df = pd.DataFrame(df\_data)  
 st.markdown(df.to\_html(escape=False, index=False), unsafe\_allow\_html=True)  
   
 # Summary statistics  
 st.markdown("---")  
 col1, col2, col3, col4 = st.columns(4)  
   
 with col1:  
 st.metric("📊 Total Results", len(st.session\_state.current\_results))  
   
 with col2:  
 if st.session\_state.current\_query\_type == "list\_facilities":  
 divisions = set(r.get('division', 'N/A') for r in st.session\_state.current\_results if r.get('division'))  
 st.metric("🏢 Divisions", len(divisions))  
 elif st.session\_state.current\_query\_type == "new\_expansions":  
 greenfield = sum(1 for r in st.session\_state.current\_results if r.get('expansion\_type') == 'greenfield')  
 st.metric("🌱 Greenfield", greenfield)  
 elif st.session\_state.current\_query\_type == "hiring\_positions":  
 factory\_roles = sum(1 for r in st.session\_state.current\_results if r.get('is\_factory\_role'))  
 st.metric("🏭 Factory Roles", factory\_roles)  
   
 with col3:  
 if st.session\_state.current\_query\_type in ["list\_facilities", "new\_expansions"]:  
 operational = sum(1 for r in st.session\_state.current\_results if r.get('status') == 'operational')  
 st.metric("✅ Operational", operational)  
 else:  
 locations = set(r.get('location', 'N/A') for r in st.session\_state.current\_results if r.get('location'))  
 st.metric("📍 Locations", len(locations))  
   
 with col4:  
 confidences = [r.get('confidence', 0) for r in st.session\_state.current\_results]  
 avg\_conf = sum(confidences) / len(confidences) if confidences else 0  
 st.metric("🎯 Avg Confidence", f"{avg\_conf:.2f}")  
   
 else:  
 # BETTER ERROR MESSAGE  
 st.error(f"""  
 🔍 \*\*No results found for query type: {st.session\_state.current\_query\_type}\*\*  
   
 \*\*Possible reasons:\*\*  
 1. No data in database matches your filters  
 2. Try Query 1 first (List All Facilities) to see what data exists  
 3. Check if ingestion pipeline completed successfully  
 4. Try removing filters in the sidebar  
   
 \*\*Debug Info:\*\*  
 - Query Type: {st.session\_state.current\_query\_type}  
 - Applied Filters: {filters}  
 """)  
   
 # Evidence viewer  
 if len(st.session\_state.current\_evidence) > 0:  
 st.markdown("---")  
 st.subheader("📄 Evidence & Citations")  
 st.caption("Click on citations in the table above to jump to highlighted evidence")  
   
 for idx, evidence in enumerate(st.session\_state.current\_evidence):  
 with st.expander(f"\*\*[{idx + 1}]\*\* {evidence.get('title', 'Evidence')} - {evidence.get('source\_type', 'Document')}"):  
 # Source metadata  
 st.markdown(f"\*\*🔗 URL:\*\* [{evidence.get('url', 'N/A')}]({evidence.get('url', '#')})")  
 st.markdown(f"\*\*📅 Date:\*\* {evidence.get('date', 'N/A')}")  
 st.markdown(f"\*\*📊 Confidence:\*\* {get\_confidence\_badge(evidence.get('confidence', 0.7))}", unsafe\_allow\_html=True)  
   
 st.markdown("\*\*📝 Evidence Snippet:\*\*")  
   
 # Display text  
 text = evidence.get('text', '')  
 st.markdown(f'<div class="evidence-box" id="ev-{idx}">{text}</div>', unsafe\_allow\_html=True)  
  
else:  
 # Initial state - welcome message  
 st.info("👆 Click on a preset query button or enter a custom query to get started")  
  
st.markdown("---")  
st.subheader("ℹ️ About this Platform")  
st.markdown("""  
This platform provides AI-powered intelligence on \*\*Motherson India's\*\* facilities, expansions, and hiring.  
  
\*\*✨ Features:\*\*  
- 🏭 \*\*Facility Intelligence\*\*: Track all India facilities across divisions  
- 🏗️ \*\*Expansion Monitoring\*\*: Identify greenfield and expansion projects with timelines  
- 👷 \*\*Hiring Insights\*\*: Surface factory-related hiring positions (excludes IT/software roles)  
- 📊 \*\*Evidence-Based\*\*: All answers backed by citations  
- 🔍 \*\*Advanced Filters\*\*: Filter by division, state, status, date range  
- 🎯 \*\*Confidence Scoring\*\*: Know how reliable each result is  
- 🇮🇳 \*\*India-Focused\*\*: Only shows Indian facilities and locations  
  
\*\*📚 Data Sources:\*\*  
- Official Motherson website (address directory with 170+ facilities)  
- Annual reports and presentations (FY 2023-24)  
- Career pages (factory job postings)  
- Public web sources  
  
\*\*🔍 Query Examples:\*\*  
- "Show all wiring facilities in Gujarat"  
- "List expansions announced in 2024"  
- "What factory jobs are available in Tamil Nadu?"  
""")  
  
# Footer  
st.markdown("---")  
st.caption("Motherson Intelligence Platform | Powered by Agentic RAG & NLP | India Focus | Data refreshed: " + datetime.now().strftime("%Y-%m-%d %H:%M"))

## src\ui\\_\_init\_\_.py

# ==================================================  
# File: src/ui/\_\_init\_\_.py  
# ==================================================  
"""  
Streamlit user interface module  
"""