Charm.li URL Processing & Embedding System - Complete Guide

Table of Contents

Overview

What This System Does

This system processes automotive repair manual URLs from charm.li for multiple vehicle brands (52 brands total). The process involves:

1. **Filtering URLs** - Extract URLs for a specific brand from the master URL file
2. **Generating Embeddings** - Create semantic vector embeddings for each URL using OpenAI's API
3. **Building FAISS Index** - Construct a compressed IVF-PQ FAISS index for fast similarity search
4. **Database Migration** - Move URL metadata to PostgreSQL to eliminate large pickle files
5. **URL Matching Service** - Deploy a memory-efficient API service for URL lookups

Why This Matters

- **Semantic Search**: Find relevant repair manuals based on meaning, not just keywords
- **Memory Efficiency**: Process millions of URLs without running out of RAM
- **Fast Retrieval**: FAISS index enables sub-second searches across millions of records
- **Scalable**: Database-backed system eliminates file-based bottlenecks

Resource Requirements

⚠️ **CRITICAL**: This process is **EXTREMELY resource-intensive**

Component	Requirements	Notes
RAM	16-64 GB	Depends on brand size (Ford: 23.5M URLs requires 32GB+)
Storage	50-200 GB per brand	Embeddings + FAISS index + checkpoints
GPU	Optional	CPU works, but GPU speeds up FAISS index building
Time	4-48 hours per brand	Depends on concurrency and API rate limits

API Costs	$50-500 per brand	OpenAI embedding API costs (1536-dim embeddings)

System Architecture

```
│  - Semantic search: "2020 Ford F150 engine diagnostic"        │
│  - Returns: Top 15 matching URLs with similarity scores       │
└──────────────────────────────────────────────────────────────┘
```

Prerequisites

Required Software

```
# Python 3.9+ (3.10 recommended)
python --version

# PostgreSQL 13+ (for database migration)
psql --version

# Required Python packages
pip install -r requirements.txt
```

Required Files

1. **Master URL File**: `all_urls_29732.txt.gz` (contains all charm.li URLs)
2. **Processing Script**: `scripts/process_urls_charmli.py`
3. **Migration Script**: `migrate_to_db.py`
4. **URL Matcher**: `core/services/chevrolet_url_matcher.py` (brand-agnostic)

Environment Variables

Create a `.env` file in the project root:

```
# OpenAI API (for embeddings)
OPENAI_API_KEY=sk-...

# PostgreSQL Database (for URL metadata)
DATABASE_URL=postgresql://user:password@localhost:5432/skilled2hire

# Optional: Async database URL
ASYNC_DATABASE_URL=postgresql+asyncpg://user:password@localhost:5432/skilled2hire
```

Step-by-Step Processing Guide

Phase 1: URL Filtering (Per Brand)

Goal: Extract URLs for a specific brand from the master file

Python Script Filtering

```
# filter_brand_urls.py
import gzip
import re
```

```
from pathlib import Path

def filter_brand_urls(brand: str, master_file: str = "all_urls_29732.txt.gz"):
    """Extract URLs for a specific brand"""
    output_file = f"{brand.lower()}_urls_logged.txt.gz"

    # Case-insensitive pattern: /brandname/
    pattern = re.compile(rf'/{re.escape(brand)}/', re.IGNORECASE)

    count = 0
    with gzip.open(master_file, 'rt', encoding='utf-8') as infile:
        with gzip.open(output_file, 'wt', encoding='utf-8') as outfile:
            for line in infile:
                if pattern.search(line):
                    outfile.write(line)
                    count += 1

    print(f"✓ Extracted {count:,} URLs for {brand}")
    print(f"✓ Saved to: {output_file}")
    return output_file

# Usage
filter_brand_urls("chevrolet")
filter_brand_urls("ford")
filter_brand_urls("bmw")
```

Supported Brands (52 total):

Phase 2: Embedding Generation & FAISS Index Building

Goal: Generate semantic embeddings and build compressed FAISS index

⚠️ **IMPORTANT**: This is the most resource-intensive step!

2.1 Check Available Resources

```
# Check available RAM
free -h

# Check available disk space
df -h .

# Estimate required resources for your brand
python -c "
import gzip
brand = 'ford'  # Change to your brand
with gzip.open(f'{brand}_urls_logged.txt.gz', 'rt') as f:
    count = sum(1 for _ in f)
print(f'URLs: {count:,}')
print(f'Estimated RAM needed: {count * 10 / 1024 / 1024:.1f} GB')
print(f'Estimated storage: {count * 6144 / 1024 / 1024 / 1024:.1f} GB')
print(f'Estimated API cost: ${count * 0.00002:.2f}')
"
```

2.2 Enable Swap Memory (if needed)

If you have insufficient RAM, enable swap space:

```
# Create 32GB swap file (adjust size as needed)
sudo fallocate -l 32G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile

# Verify swap is active
free -h

# Make permanent (optional)
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
```

2.3 Run Processing Script

```
# Basic usage
python scripts/process_urls_charmli.py --brand chevrolet

# Advanced usage with custom parameters
python scripts/process_urls_charmli.py \
    --brand ford \
    --input ford_urls_logged.txt.gz \
    --output ford_urls_processed.pkl \
    --concurrency 50 \
    --checkpoint-every 10000 \
    --resume

# For very large datasets (Ford, GM), use lower concurrency
python scripts/process_urls_charmli.py \
    --brand ford \
    --concurrency 20 \
    --checkpoint-every 5000
```

Command Line Options:

Flag	Description	Default
`--brand`	Brand name (e.g., chevrolet, ford)	chevrolet
`--input`	Input file path	`{brand}_urls_logged.txt.gz`
`--output`	Output pickle file	`{brand}_urls_processed.pkl`
`--concurrency`	API request concurrency	50
`--checkpoint-every`	Save checkpoint every N URLs	10000
`--resume`	Resume from checkpoint	True
`--no-resume`	Start fresh (ignore checkpoints)	-

2.4 Monitor Progress

The script will output detailed progress:

```
[INFO] Processing Chevrolet URLs...
[INFO] Input: chevrolet_urls_logged.txt.gz
[INFO] Output: chevrolet_urls_processed.pkl
[INFO] Found 1,234,567 URLs to process
[INFO] Starting embedding generation...
[INFO] Checkpoint: 10000/1234567 (0.8%) | Speed: 125 URLs/sec | ETA: 2h 45m
```

```
[INFO] Checkpoint: 20000/1234567 (1.6%) | Speed: 130 URLs/sec | ETA: 2h 35m
...
[INFO] Building FAISS IVF-PQ index...
[INFO] IVF-PQ parameters: nlist=1112, m=8, nbits=8
[INFO] Compression: 6144 bytes -> 8 bytes per vector (768x compression)
[INFO] Training IVF-PQ index on 284,928 samples...
[INFO] Adding 1,234,567 vectors to index...
[INFO] ✓ FAISS index built, memory used: 0.45 GB
[INFO] ✅ Preprocessing Complete!
```

2.5 Handle Interruptions

If the process crashes or you stop it (Ctrl+C):

```
# Resume from last checkpoint (automatic)
python scripts/process_urls_charmli.py --brand chevrolet --resume

# The script will detect checkpoint files:
# - chevrolet_urls_processed_embeddings.npy (partial embeddings)
# - chevrolet_urls_processed_metadata.json (completion status)
```

2.6 Verify Output Files

```
# Check output files
ls -lh chevrolet_urls_processed*

# Expected files:
# chevrolet_urls_processed.pkl          (~500 MB - 5 GB, metadata)
# chevrolet_urls_processed_embeddings.npy (~7 GB - 150 GB, vectors)
# chevrolet_urls_processed_faiss_ivfpq.index (~50 MB - 2 GB, compressed)

# Test loading the pickle
python -c "
import pickle
with open('chevrolet_urls_processed.pkl', 'rb') as f:
    data = pickle.load(f)
print(f'URLs: {len(data[\"processed_urls\"]):,}')
print(f'Embeddings shape: {data[\"embeddings_shape\"]}')
"
```

Phase 3: Database Migration (CRITICAL for Production)

Goal: Move URL metadata from pickle → PostgreSQL for memory efficiency

Why Migrate to Database?

Before (Pickle)	After (Database)
Load 1M+ URLs into RAM	Query on-demand from PostgreSQL
2-10 GB RAM per brand	~50 MB RAM per brand
Cannot scale beyond 2-3 brands	Support all 52 brands simultaneously
Slow startup (10-30 seconds)	Instant startup (<1 second)

3.3 Migrate URL Metadata

```
# Migrate a single brand
python migrate_to_db.py --brand chevrolet

# Migrate with options
python migrate_to_db.py \
    --brand ford \
    --batch-size 50000 \
    --memory-limit 8

# Clear existing data and migrate (fresh start)
python migrate_to_db.py --brand chevrolet --clear --yes

# Migrate ALL brands (if you have enough time)
python migrate_to_db.py --brand all --clear --yes
```

Command Line Options:

Flag	Description	Default
`--brand`	Brand name or "all"	required
`--batch-size`	Insert batch size	50000
`--memory-limit`	RAM limit in GB	2.0
`--clear`	Clear existing data first	False
`--yes`	Skip confirmation prompts	False
`--delete-files`	Delete pickle/npy after migration	False

3.4 Monitor Migration Progress

```
[INFO] MIGRATING BRAND: CHEVROLET
[INFO] Pickle file size: 2.35 GB
[INFO] Found 1,234,567 URLs to migrate
[INFO] Using FAST PostgreSQL COPY method...
[INFO] Processing: 0 to 50,000 (4.0%)
[INFO]   Inserted 50,000 records in 3.2s (15,625 rec/sec)
[INFO]   Overall: 50,000/1,234,567 | Speed: 15,625 rec/sec | ETA: 1.3 min
[INFO] Processing: 50,000 to 100,000 (8.1%)
...
[INFO] SUCCESS: Successfully migrated 1,234,567 URLs for chevrolet
[INFO] SUCCESS: Verified: 1,234,567 records in database
```

3.5 Verify Migration

```
# Check database records
psql -U skilled2hire_user -d skilled2hire -c "
SELECT brand, COUNT(*) as url_count
FROM brand_url_metadata
GROUP BY brand
ORDER BY url_count DESC;
"

# Sample output:
```

```
# Sample output:
#    brand    | url_count
# ------------+-----------
#  ford       | 23,500,000
#  chevrolet  |  1,234,567
#  toyota     |    985,234
```

3.6 Cleanup (Optional)

If migration succeeded and you want to save disk space:

```
# Delete pickle and embeddings files (CAUTION!)
python migrate_to_db.py --brand chevrolet --delete-files --yes

# NOTE: Keep the FAISS index file! It's required for searches
# DO NOT DELETE: chevrolet_urls_processed_faiss_ivfpq.index
```

Automation Options

Why NOT Fully Automate?

1. **Resource Constraints**: Processing all 52 brands requires 200-500 GB RAM + 5 TB storage
2. **Error Handling**: Each brand has unique edge cases sometimes (special characters, encoding)

Semi-Automated Approach (Recommended)

Strategy 1: Queue-Based Processing

Process brands sequentially with a simple queue system:

```
# process_queue.py
import subprocess
import time
from pathlib import Path

# Priority queue (most commonly used brands first)
PRIORITY_BRANDS = [
    "chevrolet", "ford", "gmc", "dodge", "ram",
    "toyota", "honda", "nissan", "bmw", "volkswagen"
]

SECONDARY_BRANDS = [
    "audi", "mercedes-benz", "porsche", "subaru", "mazda",
    "kia", "hyundai", "jeep", "chrysler", "buick"
]

def process_brand(brand: str, data_dir: Path):
    """Process a single brand (filtering → embedding → migration)"""
    print(f"{'='*60}")
    print(f"PROCESSING BRAND: {brand.upper()}")
    print(f"{'='*60}")

    try:
        # Step 1: Filter URLs (if not already done)
        url_file = data_dir / f"{brand}_urls_logged.txt.gz"
        if not url_file.exists():
            print(f"⚠️  Missing URL file: {url_file}")
```

```
            print(f"   Please filter URLs first: grep -i '/{brand}/' all_urls_29732.txt.gz")
            return False

        # Step 2: Generate embeddings
        print(f"\n[1/3] Generating embeddings...")
        result = subprocess.run([
            "python", "scripts/process_urls_charmli.py",
            "--brand", brand,
            "--concurrency", "30",
            "--resume"
        ], capture_output=False, text=True)

        if result.returncode != 0:
            print(f"✗ Failed to process {brand}")
            return False

        # Step 3: Migrate to database
        print(f"\n[2/3] Migrating to database...")
        result = subprocess.run([
            "python", "migrate_to_db.py",
            "--brand", brand,
            "--yes"
        ], capture_output=False, text=True)

        if result.returncode != 0:
            print(f"✗ Failed to migrate {brand}")
            return False

        print(f"\n✓ Successfully processed {brand}")
        return True

    except Exception as e:
        print(f"✗ Error processing {brand}: {e}")
        return False

def main():
    data_dir = Path.cwd()

    # Process priority brands first
    for brand in PRIORITY_BRANDS:
        success = process_brand(brand, data_dir)
        if not success:
            print(f"\n⚠️  Pausing queue due to error with {brand}")
            print(f"   Fix the issue and restart from: {brand}")
            break

        # Sleep between brands (avoid API rate limits)
        print(f"\n😴 Cooling down for 5 minutes...")
        time.sleep(300)

    print(f"\n{'='*60}")
    print(f"QUEUE COMPLETE")
    print(f"{'='*60}")

if __name__ == "__main__":
    main()
```

Run the queue:

```
# Start processing (runs until error or completion)
python process_queue.py

# Monitor in another terminal
watch -n 5 'ls -lh *_urls_processed* | tail -20'
```

Strategy 2: Manual Batch Processing

Process brands in small batches (recommended for stability):

Week 1: High-Priority Brands (5 brands)

```
python scripts/process_urls_charmli.py --brand chevrolet && \
python migrate_to_db.py --brand chevrolet --yes && \
python scripts/process_urls_charmli.py --brand ford && \
python migrate_to_db.py --brand ford --yes && \
python scripts/process_urls_charmli.py --brand toyota && \
python migrate_to_db.py --brand toyota --yes
```

Week 2: Medium-Priority Brands (10 brands)

```
for brand in honda nissan bmw volkswagen audi mazda kia hyundai jeep dodge; do
    echo "Processing $brand..."
    python scripts/process_urls_charmli.py --brand $brand && \
    python migrate_to_db.py --brand $brand --yes
    echo "Cooling down..."
    sleep 300
done
```

Strategy 3: API-Based Processing Service

For true automation, create a background worker service:

```
# worker_service.py
from celery import Celery
from pathlib import Path
import subprocess

app = Celery('url_processor', broker='redis://localhost:6379/0')

@app.task(bind=True, max_retries=3)
def process_brand_task(self, brand: str):
    """Celery task for processing a brand"""
    try:
        # Run processing script
        result = subprocess.run([
            "python", "scripts/process_urls_charmli.py",
            "--brand", brand,
            "--resume"
        ], capture_output=True, text=True, timeout=86400)  # 24h timeout

        if result.returncode != 0:
            raise Exception(f"Processing failed: {result.stderr}")

        # Run migration
        result = subprocess.run([
            "python", "migrate_to_db.py",
            "--brand", brand,
            "--yes"
        ], capture_output=True, text=True, timeout=3600)  # 1h timeout

        if result.returncode != 0:
            raise Exception(f"Migration failed: {result.stderr}")
```

```
        return {"status": "success", "brand": brand}

    except Exception as e:
        # Retry with exponential backoff
        raise self.retry(exc=e, countdown=60 * (2 ** self.request.retries))

# Submit tasks
from worker_service import process_brand_task

for brand in ["chevrolet", "ford", "toyota"]:
    process_brand_task.delay(brand)
```

Troubleshooting

Common Issues

Issue 1: Out of Memory (OOM)

Symptoms:

```
MemoryError: Unable to allocate array
Killed (process terminated)
```

Solutions:

1. Enable swap memory (see Phase 2.2)
2. Reduce `--concurrency` flag
3. Increase `--checkpoint-every` to save more frequently
4. Use a machine with more RAM

```
# Check memory usage
free -h
htop

# Enable swap
sudo fallocate -l 32G /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

Issue 2: OpenAI API Rate Limits

Symptoms:

```
openai.error.RateLimitError: Rate limit exceeded
```

Solutions:

1. Reduce `--concurrency` (default: 50 → try 20-30)
2. The script has built-in backoff, just wait

3. Check your OpenAI API quota

```
# Retry with lower concurrency
python scripts/process_urls_charmli.py --brand chevrolet --concurrency 20
```

Issue 3: Checkpoint Files Corrupt

Symptoms:

```
ValueError: Cannot load checkpoint metadata
```

Solutions:

```
# Remove checkpoint files and start fresh
rm chevrolet_urls_processed_embeddings.npy
rm chevrolet_urls_processed_metadata.json
rm chevrolet_urls_processed_urls.pkl.gz

# Restart processing
python scripts/process_urls_charmli.py --brand chevrolet --no-resume
```

Issue 4: FAISS Index Building Fails

Symptoms:

```
RuntimeError: Error in faiss::IndexIVFPQ::train
```

Solutions:

1. Not enough training samples (need at least 256 per cluster)
2. Dimension mismatch (embeddings must be 1536-dim)

```
# Verify embeddings file
python -c "
import numpy as np
embeddings = np.load('chevrolet_urls_processed_embeddings.npy')
print(f'Shape: {embeddings.shape}')
print(f'Expected: (N, 1536)')
"
```

Issue 5: Database Migration Too Slow

Symptoms:

```
[INFO] Speed: 50 rec/sec | ETA: 6 hours
```

Solutions:

Solutions:

1. The script uses PostgreSQL COPY (fastest method)
2. Check database connection latency
3. Increase `--batch-size`

```
# Check database connection
psql -U skilled2hire_user -d skilled2hire -c "SELECT 1"

# Use larger batch size
python migrate_to_db.py --brand chevrolet --batch-size 100000
```

Technical Details

FAISS Index Types

The system supports multiple FAISS index types:

Index Type	Compression	Speed	Accuracy	Memory
IVF-PQ (default)	4-32x	Fast	Good (90-95%)	Low
HNSW	None	Very Fast	Excellent (98%)	Medium
Flat	None	Fast	Perfect (100%)	High

IVF-PQ Parameters (auto-tuned):

- `nlist`: Number of clusters = `sqrt(num_urls)` (max 4096)
- `m`: Subquantizers = 8 (divides 1536 evenly)
- `nbits`: Bits per subquantizer = 8 (256 centroids)
- Result: **768x compression** (6144 bytes → 8 bytes per vector)

Database Schema

```
CREATE TABLE brand_url_metadata (
    id SERIAL PRIMARY KEY,
    brand VARCHAR(50) NOT NULL,
    faiss_index INTEGER NOT NULL,
    url TEXT NOT NULL,
    path TEXT,
    year INTEGER,
    model VARCHAR(255),
    extra_metadata JSONB,
    created_at TIMESTAMPTZ DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    CONSTRAINT uq_brand_faiss_index UNIQUE (brand, faiss_index)
);

CREATE INDEX idx_brand_url_metadata_brand ON brand_url_metadata(brand);
CREATE INDEX idx_brand_url_metadata_year ON brand_url_metadata(brand, year);
CREATE INDEX idx_brand_url_metadata_model ON brand_url_metadata(brand, model);
```

URL Parsing Logic

URL Parsing Logic

The system extracts structured metadata from URLs:

```
URL: /chevrolet/2020-silverado-1500/engine/diagnostic-trouble-codes

Parsed:
  brand: chevrolet
  year: 2020
  model: silverado-1500
  category: engine
  subcategory: diagnostic-trouble-codes
  path: /chevrolet/2020-silverado-1500/engine/diagnostic-trouble-codes
```

Memory Optimization Techniques

1. **Memory-Mapped Embeddings**: Load embeddings on-demand (no RAM spike)
2. **Streaming Checkpoints**: Write embeddings directly to disk
3. **Database-Backed Metadata**: Query URLs from PostgreSQL (not RAM)
4. **FAISS IVF-PQ**: 768x compression reduces index size
5. **Chunked Processing**: Process URLs in batches (10K-50K)

Cost Estimation

OpenAI Embedding API Costs (per brand):

Brand	URLs	Embeddings Cost	Processing Time
Chevrolet	1.2M	$24	4-8 hours
Ford	23.5M	$470	40-48 hours
Toyota	985K	$19.70	3-6 hours
BMW	650K	$13	2-4 hours
Total (52 brands)	~50M	**~$1,000**	**~15 days**

Pricing based on OpenAI's text-embedding-3-large @ $0.00002/1K tokens

---

Quick Reference Cheat Sheet

Filter URLs for Brand

```
zcat all_urls_29732.txt.gz | grep -i "/brandname/" | gzip > brandname_urls_logged.txt.gz
```

Generate Embeddings & FAISS Index

```
python scripts/process_urls_charmli.py --brand brandname --resume
```

Migrate to Database

```
python migrate_to_db.py --brand brandname --yes
```

Check Progress

```
# Files
ls -lh brandname_urls_processed*

# Database
psql -U skilled2hire_user -d skilled2hire -c \
  "SELECT COUNT(*) FROM brand_url_metadata WHERE brand='brandname'"
```

Use URL Matcher

```
from core.services.chevrolet_url_matcher import get_brand_url_matcher

matcher = get_brand_url_matcher(brand="brandname", use_database=True)
results = matcher.search_similar_urls(
    queries=["2020 model diagnostic"],
    top_k=15
)
```

Document Version: 1.0
Last Updated: January 2026
Maintained By: TheAgentic Engineering Team