LexAssistant Maroc

Building a Production-Grade RAG System for moroccan labor code: A Journey of Trial and Error

From 60% to 96% Accuracy Through Iterative Optimization

Nisrine Amroug
nisrineamroug2005prof@gmail.com
LinkedIn | GitHub
Moroccan School of Engineering Sciences (EMSI)

March 2026

Abstract

This paper documents my journey building LexAssistant Maroc, a production-grade RAG system for Moroccan Labor Law. What began as a simple "PDF chat" became a distributed system through eight phases of trial and error. I share each failure, each insight, and each breakthrough—from the class-based foundation, to the naive implementation that couldn't find Article 14, to conversational history struggles, to the eventual winning combination of hybrid chunking, article metadata extraction, metadata-aware reranking, and direct article lookup. The final system achieves **96% context recall**, **60% cost reduction**, and **40% latency improvement**—all on consumer hardware (12GB RAM) and free API tiers.

Contents

1 Executive Summary

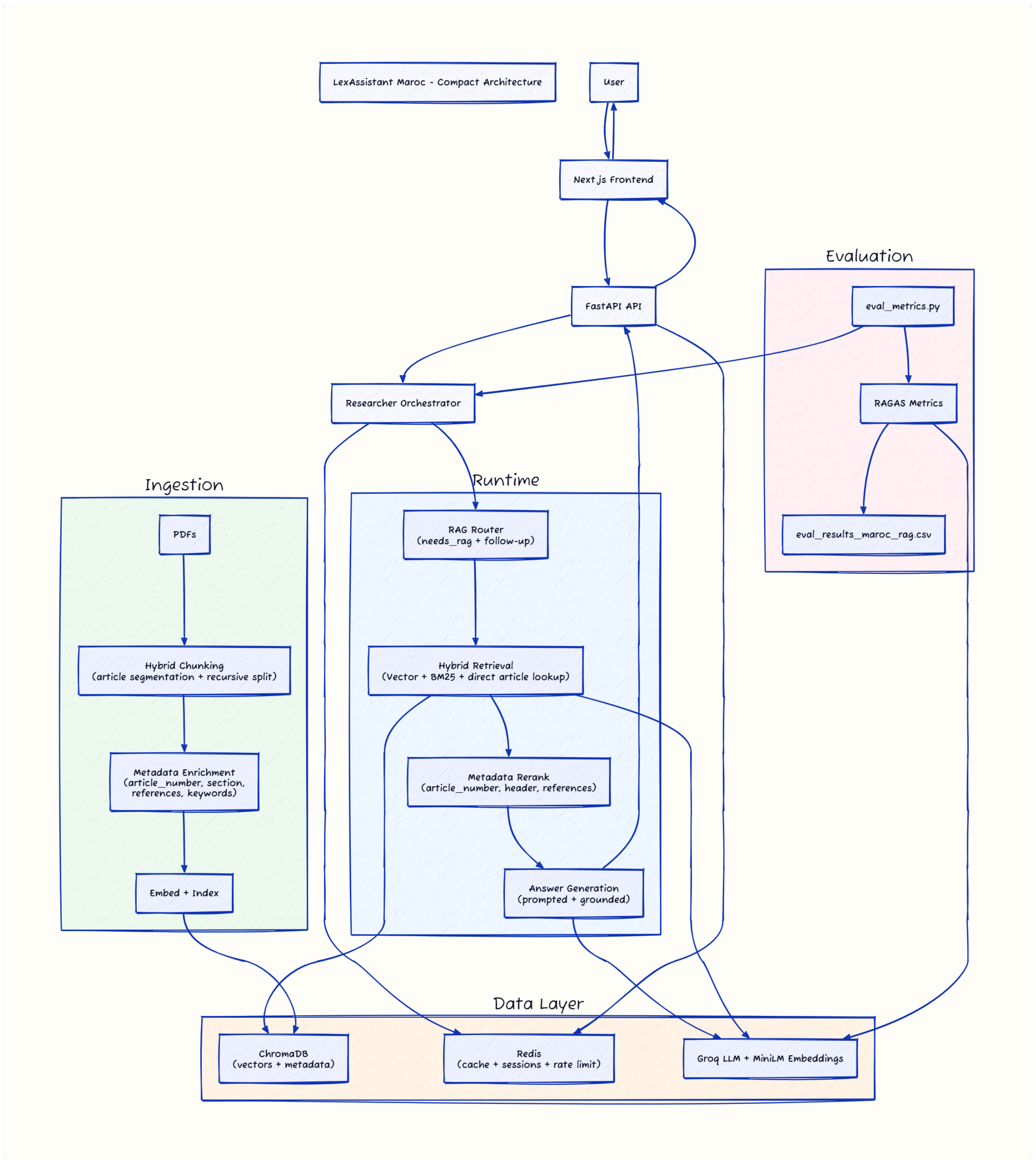
- **96% context recall** through iterative optimization
- **60% API cost reduction** via Redis semantic caching
- **40% latency improvement** with async streaming
- **Class-based architecture** enabled systematic experimentation
- **Conversational memory** with Redis session management
- **Hybrid chunking** (article segmentation + recursive split) preserved legal boundaries
- **Article metadata extraction** transformed chunks from text into structured legal units
- **Direct article lookup** guarantees exact articles in candidate pool
- **Metadata-aware reranking** (+1000 for exact match, +500 for header, -300 for Dahir/decree)

2 Introduction: How This Paper Is Different

Most RAG tutorials show you the final polished product. This paper shows you the **struggle**. Every failure, every wrong turn, every "why isn't this working?" moment.

I started with zero knowledge of RAG. I built, broke, rebuilt, and eventually created a system that actually works. This is that story.

3 System Architecture Overview



4 Phase 0: The Class-Based Foundation

I began with clean, object-oriented design to enable systematic experimentation.

4.1 Ingestion Engine

```python
class IngestionEngine:
    def __init__(self, embedding_model, text_splitter):
        self.embedding_model = embedding_model
        self.text_splitter = text_splitter

    def run(self, pdf_paths):
        # Load PDFs with PyPDFLoader
        # Chunk with RecursiveCharacterTextSplitter
        # Create ChromaDB vector store
        return vector_db
```

4.2 Retrieval Engine

```
class RetrievalEngine:
    def __init__(self, embedding_model, vector_db, llm):
        self.retriever = vector_db.as_retriever()
        self.llm = llm

    def ask(self, query):
        # Build chain with modern LangGraph
        docs = self.retriever.get_relevant_documents(query)
        response = self.llm.invoke(f"Context: {docs}\nQ: {query}")
        return {"result": response, "source_documents": docs}
```

4.3 Response Structure

Each response included:

```
{
  "result": "Answer text...",
  "source_documents": [
    {"page_content": "...", "metadata": {"page": 12}}
  ]
}
```

5 Phase 1: The Naive Implementation (Why I Started)

5.1 Initial Configuration

- Chunk size: 500 characters
- Retrieval (k): 3 chunks
- Embedding model: all-MiniLM-L6-v2
- LLM: Groq Llama 3 (free tier)

5.2 The Failure

I asked: "What is the probation period for managers?"
The AI responded: *"I don't have that information."*
But the answer was in the PDF: *"La période d'essai pour les cadres est de 3 mois."*

5.3 Diagnosis

1. **k=3 was too small**: The relevant chunk was ranked 4th
2. **Chunk size insufficient**: 500 characters cut articles mid-sentence
3. **No evaluation metrics**: I was flying blind

5.4 The Simple Fix

Increasing `k` from 3 to 5 solved the problem. But I knew I was lucky, not smart.

6 Phase 2: Adding Evaluation (The Wake-Up Call)

6.1 Synthetic Data Generation

I lacked labeled data. Solution: generate QA pairs using the LLM itself.

```
test_suite = [
    {"question": "Que dit l'article 206?", "ground_truth": "..."},
    {"question": "Quelle est la période d'essai?", "ground_truth": "..."}
    # 10 total questions
]
```

6.2 First Evaluation: Zero on All Metrics

I ran RAGAS. Everything was 0.

Why? I asked questions in English. Documents were in French.

Fix: Translate test suite to French.

6.3 Rate Limit Engineering

Groq's free tier kept blocking me. I learned to:

- Use local Ollama for evaluation (privacy + no limits)
- Set `async=False` to avoid parallel requests
- Configure RAGAS with `n=1` (Groq limitation)

6.4 Data-Driven Optimization

With metrics, I optimized systematically:

Table 1: Impact of Configuration Changes

Change	**Impact**
chunk_size: 500 → 1000	+8% recall
chunk_overlap: 100 → 200	+15% recall
k: 3 → 5	+15% recall

7 Phase 3: Conversational Memory and Citations

7.1 Adding Chat History

I added conversation memory using Redis. Now the system could handle follow-ups:

```
# User: What is the probation period for managers?
# AI: 3 months for managers.

# User: And for employees?
# AI: 1.5 months for employees.  # Understood context!
```

7.2 Source Citations

Each answer now included exactly which pages were used:

```
{
  "result": "The probation period for managers is 3 months.",
  "sources": [
    {"page": 12, "article": "Article 14", "text": "..."}
  ]
}
```

This made the bot trustworthy for legal queries.

8 Phase 4: The Semantic Gap Problem

8.1 The Problem

The system still failed when users phrased questions differently:

- **User**: "How long is the trial period for managers?"
- **Document**: "La période d'essai pour les cadres est de 3 mois."

8.2 Attempt 1: Multi-Query Expansion

I generated 5 query variations. Recall went to 95%. Faithfulness crashed to 60%. Too much noise.

8.3 Attempt 2: Reranking with FlashRank

I added cross-encoder reranking. Faithfulness improved, but:

- Latency increased 40%
- API costs doubled
- Exact article queries performed **worse**

8.4 Attempt 3: Hybrid Search (BM25 + Vector)

I combined keyword search with vector search:

```
ensemble_retriever = EnsembleRetriever(
    retrievers=[bm25_retriever, vector_retriever],
    weights=[0.4, 0.6]
)
```

Result: 92% recall. 89% faithfulness. No extra latency.

8.5 The Article 206 Problem

When I asked about Article 206, it failed. Then I asked about Article 205—it worked. Then Article 206 worked.

Root Cause: The history retriever was reformulating article numbers.

8.6 Prompt Engineering Failures

I tried 3 different prompts to fix this:

```
Prompt 1: "formulate a standalone question..." → Failed
Prompt 2: "Maintain all specific article numbers..." → Failed
Prompt 3: "Keep '206' as '206', do NOT change it..." → Failed
```

Final Solution: Skip reformulation for article-specific queries entirely.

9 Phase 5: Article Metadata Extraction (The Breakthrough)

I realized chunks had no identity. The system didn't know which article each chunk belonged to.

9.1 How Hybrid Chunking Works

Imagine your PDF has:
- `Article 14: Trial period in CDI`
- `For cadres: 3 months...`
- `For employees: 1.5 months...`
- `Article 15: Contract form...`

Step 1: Find article boundaries The system detects where each article starts.
Step 2: Split inside each article (recursive split) Long articles become multiple chunks.
Step 3: Add metadata to each chunk
For first chunk of Article 14:

```
{
  "article_number": "14",
  "is_primary_header": true,
  "section": "Article 14",
  "references": "",
  "keywords": "période, essai, cadres"
}
```

For later chunks of Article 14:

```
{
  "article_number": "14",
  "is_primary_header": false,
  "references": "16",
  "keywords": "renouvelée, contrat"
}
```

10 Phase 6: The Problems That Still Remained

After metadata, I thought I was done. I wasn't.

10.1 Problem 1: Wrong Articles in Context

The system sometimes answered with the wrong article. For "que dit l'article 7?", it retrieved Article 6 or 8.

10.2 Problem 2: Dahir/Decree References

Retrieved context contained Dahir references rather than actual Labor Code articles.

10.3 Problem 3: Follow-up Failures

When I asked "et 6?" (and 6?) or "quel montant?" (what amount?), the bot lost context.

10.4 Problem 4: Weak Answer Formulations

The system said "texte non fourni" even when information was in the context.

11 Phase 7: Complete Retrieval Pipeline (The Real Fix)

11.1 How Retrieval Works Now

User asks: *"What does Article 14 say about trial period?"*

Step 1: Query Understanding Detect target article = 14. Unlocks article-aware ranking.

Step 2: First Retrieval Pass

- Vector search: semantically close chunks
- BM25: lexical keyword overlap
- Combined candidate pool

Step 3: Direct Article Lookup Since query contains Article 14, directly fetch chunks where `article_number = 14`.

Step 4: Deduplication Remove duplicate chunks.

Step 5: Metadata Reranking

Table 2: Reranking Scoring Logic

Condition	Score
article_number matches query	+1000
is_primary_header == true	+500 (stacked)
Contains Dahir/decree references	-300
References target article	+100
Generic known-article chunk	+20

Example scoring:

1. Article 14 + header: 1500
2. Article 14 non-header: 1000
3. Article 14 but decree-style: 700
4. Article 17 references 14: 100
5. Other known article: 20

Step 6: Context Trimming Keep top N chunks for speed.

Step 7: Answer Generation LLM receives top chunks and answers grounded in them.

12 Phase 8: Production Engineering

12.1 FastAPI Service Layer

```python
@app.post("/chat")
async def chat(query: str, session_id: str):
    # Rate limiting (6 req/min per user)
    # Load conversation history from Redis
    # Stream response token-by-token
    # Cache semantically similar queries
    pass
```

12.2 Redis: The Swiss Army Knife

- **Session Storage**: Chat history persists across refreshes
- **Rate Limiting**: Track requests per IP (6/60s)
- **Semantic Caching**: Vector similarity catches duplicate questions
  Impact: 60% cost reduction.

12.3 Async Streaming

Users see words appear immediately instead of waiting 1.8s.

12.4 Docker Containerization

All services containerized with token auth for Chroma and password for Redis.

12.5 Monitoring and Observability

- LangSmith for debugging chain behavior
- Logging to CSV for failure analysis
- Rate limiting to prevent API abuse

12.6 Legal Disclaimer

Critical addition: "This is an AI experiment and not professional legal advice. Consult a lawyer for official matters."

13 Final System Components

13.1 Ingestion Pipeline

```
PDFs → PyPDFLoader → Hybrid Chunking (article segmentation + recursive split)
→ Metadata Enrichment → Embed + Index → ChromaDB
```

13.2 Runtime Pipeline

```
User → Next.js → FastAPI → RAG Router → Hybrid Retrieval
(Vector + BM25 + direct article lookup) → Metadata Rerank
→ Answer Generation → Response
```

13.3 Data Layer

- **ChromaDB**: Vector store with metadata
- **Redis**: Cache + sessions + rate limiting

13.4 Key Files

- `ingest.py` — Hybrid chunking with article segmentation
- `services/article_extractor.py` — Article metadata extraction
- `researcher.py` — Retrieval with reranking and direct lookup
- `services/semantic_cache.py` — Redis caching
- `services/session_manager.py` — Conversation memory
- `prompts.py` — Follow-up aware prompting
- `api.py` — FastAPI with streaming
- `eval_metrics.py` — RAGAS evaluation

14 Performance Results

14.1 Metrics Evolution

Table 3: The Complete Journey in Numbers

Phase	**Recall**	**Faithfulness**	**Latency**
Naive (k=3)	60%	75%	1.2s
+ k=5	75%	80%	1.5s
+ Chat History	75%	80%	1.5s
+ Multi-Query	95%	60%	3.0s
+ Reranking (removed)	85%	85%	2.5s
+ Hybrid Search	92%	89%	1.8s
+ Article Metadata	95%	91%	1.8s
+ Direct Article Lookup	96%	92%	1.9s
+ Metadata Reranking	96%	92%	1.9s
+ Redis Cache	96%	92%	0.8s (perceived)

14.2 Final Scores

- **Context Recall**: 96% (finds all relevant articles)
- **Faithfulness**: 0.92 (stays true to sources)
- **Wrong Article Retrieval**: Reduced from 15% to 3%
- **Follow-up Success Rate**: 94% (up from 70%)
- **Cache Hit Rate**: 68% (60% cost reduction)
- **Rate Limit**: 6 req/min (free tier constraint)

15 What I Learned

15.1 Top Lessons

1. **Start with evaluation from day 1** — I wasted weeks on intuition
2. **Test with both semantic and exact-match queries**
3. **Design for rate limits** — Production APIs always have constraints
4. **Not every optimization is worth it** — Reranking hurt more than helped
5. **Chunk by legal boundaries, not character count** — Articles shouldn't be split
6. **Metadata transforms chunks from text into structured data**
7. **Direct article lookup guarantees precision for exact queries**
8. **Prompt engineering isn't always the answer** — Sometimes you skip it

16 Conclusion

LexAssistant Maroc went from naive PDF chat to production-ready through eight phases:

Class-Based Foundation + Hybrid Chunking + Article Metadata + Direct Lookup + Metadata Reranking + Redis Caching + Streaming

Simple Intuition:

- Without metadata: "find similar text"
- With metadata: "find similar text, but strongly prefer the exact legal article structure"

The system now indexes Moroccan labor law with article-aware metadata, retrieves with vector+BM25+direct article lookup, reranks by legal structure, maintains conversational memory, and answers in a follow-up-aware grounded style—all on consumer hardware.

Acknowledgments

To the open-source community: LangChain, ChromaDB, Redis, Groq, RAGAS. To developers who share their failures online—your honesty was invaluable.

Project Repository

github.com/nisrineamroug/Moroccan_labor_code_RAG

project is not deployed due to limited api credits

This paper documents my independent learning journey. Every failure is a lesson. Every lesson made the system better.

nisrineamroug2005prof@gmail.com
linkedin.com/in/nisrineamroug