RAGGAE: A multipurpose local RAG system for Adservio

Retrieval-**A**ugmented **G**eneration **G**eneralized **A**rchitecture for **E**nterprise

Olivier Vitrac, PhD., HDR | olivier.vitrac@adservio.fr – 2025-11-05

Summary

This early note discusses the design of a **generic RAG/embeddings library** can serve **CVs, reports, and tenders**, which relies on different *document adapters* using a shared *semantic core* (retrieval + re-rank + annotation + scoring). A **hybrid (dense+sparse) + cross-encoder** is proposed. The POC adds **domain-tuning** and **Natural Lalnguage interface (NLI) checks**, and is designed from day one for **traceability** (provenance spans, scores, reasons). The whole system is designed to run on minimal infrastructure: fully local MVP – GPU with 8 GB VRAM and possibly running on CPU.

> *The project RAGGAE is now mature and is available as an Adservio GitHub project. All details available in README.md. The POC can be launched as:* `uvicorn RAGGAE.cli.demo_app:app --host 0.0.0.0 --port 8000 --reload`
>
> *Access to all files, read this file in PDF.*

---

1 | Technical Review

1.1 | Embedding options (and when to use which)

A. Dense text embeddings (bi-encoders) — default for RAG

- **General English/Multilingual:** E5-family, GTE-family, bge-family, Jina, Sentence-Transformers (MiniLM, MPNet), Cohere, OpenAI, etc.
- **Pros:** fast, scalable, cheap to store/query; perfect for “retrieve top-k chunks.”
- **Cons:** retrieval scores are approximate; for high-precision ranking add a re-ranker.

B. Cross-encoders (re-rankers) — for precision at the top

- BERT/DeBERTa/Modern LLM cross-encoders (e.g., *ms-marco*-tuned) that score (query, passage) jointly.
- **Use:** take the top 50–200 dense hits, re-rank to get very accurate top-10.
- **Trade-off:** slower and costlier per query, but best quality for tenders.

C. Hybrid retrieval (dense + sparse) — when vocabulary matters

- Combine **BM25 / SPLADE** (sparse, exact terms) with dense vectors (semantics).
- **Use:** tenders have jargon, acronyms, legal clauses—hybrid boosts recall on rare terms.

D. Domain-tuned embeddings — when your domain dominates

- Light fine-tuning (or adapters) using your **historic tenders, SoWs, CVs, past responses**.
- **Use:** improves intent matching on “DevOps/MLOps” specifics, vendor boilerplate, compliance phrasing.

E. Multilingual & French

- Choose a **multilingual** model (FR/EN at minimum). If not, keep separate indices per language and route queries.
- Consider **language-aware chunking** and **query translation** as a fallback.

F. Long-document strategies (tenders/CVs/reports)

- **Hierarchical embeddings:** section → paragraph → sentence; route queries to the right level.
- **Layout-aware chunking:** keep tables, bullets, headers/footers; preserve section numbers and annex links.

1.2 | "Semantic analysis" we'll want beyond embeddings

We think of this as *signals* layered on top of retrieval:

- **Document structure parsing:** title, sections, annexes, tables, numbered requirements (MUST/SHALL/DOIT).
- **Keyphrase & requirement mining:** extract capabilities (e.g., "K8s, ArgoCD, MLflow, ISO 27001, on-prem"), constraints (SLA, RPO/RTO, sovereignty).
- **NER & taxonomy mapping:** map entities/skills/standards to an **Adservio capability ontology** (DevOps, MLOps, Security, Cloud, Data).
- **Entailment/NLI checks:** "Does our offer satisfy clause 4.2.3?" (Yes/No/Partial + rationale).
- **De-duplication & canonicalization:** normalize synonyms ("GPU farm" ≈ "on-prem compute with NVIDIA A-series").
- **Risk & eligibility flags:** deadlines, mandatory certifications, exclusion criteria, IP/sovereignty clauses.

These features feed your **scoring/ranking** (fit, risk, attractiveness) and later your **form pre-fill**.

1.3 | Can one library handle CVs, reports, tenders? (Yes—if you design it right)

Design a **document-agnostic semantic layer** with adapters:

- **Core abstractions:**
  - `Document` (metadata + pages + spans + tables)
  - `Chunk` (text, layout anchors, section path)
  - `EmbeddingProvider` (pluggable: dense, sparse, hybrid)
  - `Indexer/Retriever` (vector DB + BM25)
  - `Reranker` (cross-encoder)
  - `Annotator` (NER, keyphrases, taxonomy linker)
  - `Scorer` (tender-fit, confidence, risk)
  - `Extractor` (field mappers for pre-fill)
- **Adapters per doc type:** `TenderAdapter`, `CVAdapter`, `ReportAdapter` implement:
  - **Parsing rules** (e.g., numbered requirements vs. experiences vs. results)
  - **Chunking rules** (keep bullets, tables, job periods)
  - **Field mappers** (e.g., "Lot 2 scope" → `scope.devops`, "Years exp in K8s"

→ cv.skills.k8s.years)

Result: *same* embedding/retrieval engine, *different* adapters and scoring logic.

1.4 | Minimal technical blueprint

```
class EmbeddingProvider:
    def embed_texts(self, texts: list[str]) ->
list[list[float]]NumPy OK: 1.26.4
ST OK. dim: 384: ...
    def embed_query(self, text: str) -> list[float]: ...

class DenseBiEncoder(EmbeddingProvider): ...
class SparseBM25: ...
class HybridRetriever:
    def __init__(self, dense: EmbeddingProvider, sparse:
SparseBM25, alpha=0.6): ...
    def search(self, query: str, k=100) -> list["Hit"]: ...

class CrossEncoderReranker:
    def rerank(self, query: str, hits: list["Hit"], top_k=20) -
> list["Hit"]: ...

class DocumentAdapter:
    def parse(self, raw_bytes) -> "Document": ...
    def chunk(self, doc: "Document") -> list["Chunk"]: ...
    def annotate(self, chunks) -> list["Chunk"]: ...
    def score(self, query, chunks) -> list["ScoredChunk"]: ...

# Pipeline
adapter = TenderAdapter(lang="fr")
doc = adapter.parse(pdf_bytes)
chunks = adapter.chunk(doc)
vectors = dense.embed_texts([c.text for c in chunks])
index.upsert(chunks, vectors, metadata=adapter.annotations)

hits = hybrid.search(query, k=150)
hits = reranker.rerank(query, hits, top_k=25)
```

1.5 | Choosing an embedding setup (quick decision guide)

- **Early phase / fast demo:** Multilingual dense bi-encoder + BM25 hybrid; add a small cross-encoder re-ranker.
- **Production quality for tenders:** Same as above **plus** (a) domain-tuning on historical tenders & responses, (b) taxonomy-aware scoring, (c) NLI compliance checks.
- **High privacy / on-prem:** Prefer **open models** (no external API), self-host vector DB (FAISS, Qdrant, Milvus).
- **Strict FR/EN mix:** Multilingual embeddings *or* per-language indices with automatic routing.
- **Lots of tables/forms:** Ensure **layout-aware parsing** (tables become key-value triples; keep cell coordinates).

Absolutely feasible locally: E5-small + BM25 + optional cross-encoder, FAISS index, Ollama (7–8B Q4) for NLI/extraction.

One generic library with adapters lets you handle tenders, CVs, and reports with the same semantic core.1.6 | Ranking & classification for tenders

- **Relevance ranking:** Hybrid retrieve → cross-encode re-rank.
- **Fit scoring:** weighted signals (must-haves met, certifications present, tech match, budget window, delivery window, jurisdiction).
- **Classification buckets:** DevOps/MLOps/Lot-based labels via:
  - **Zero-shot** (NLI prompt + label descriptions) for cold start.
  - **Few-shot supervised** (logistic regression or small classifier on embeddings) once you have labeled data.
  - **Topic modeling** (BERTopic/Top2Vec on embeddings) for discovery of recurring themes.

1.7 | Toward pre-filling response forms (step 2)

- **Field schema registry:** define each target field with a canonical name, regex/ontology, and examples.
- **Extractor chain:** retrieval → NER/regex → NLI validation → *LLM with constrained generation* to map spans to fields.
- **Traceability:** keep source spans + page numbers (for audit and human review).
- **Safety gates:** mandatory fields coverage, confidence thresholds, red-flag clauses (IP/sovereignty/insurance).

1.8 | Evaluation of prototype from day 1

- **Retrieval:** Recall@k, nDCG on a seed set of queries (FR/EN).
- **Re-ranking:** MRR@10, precision@5.
- **Classification:** F1 per class, macro-F1; calibration curve.
- **Extraction (pre-fill):** exact-match / relaxed-match and **provenance coverage** (% fields with verified source span).
- **Human-in-the-loop:** review time saved per tender.

1.9 | Practical shortlist (safe bets to prototype)

- **Dense bi-encoder:** a strong multilingual Sentence-Transformers-style model (or equivalent GTE/bge multilingual).
- **Sparse:** BM25; consider SPLADE later if needed.
- **Re-ranker:** MS-MARCO-style cross-encoder or a modern cross-encoder fine-tuned on your domain pairs.
- **Vector DB:** FAISS (embedded) → Qdrant/Milvus (server) when scaling.
- NumPy OK: 1.26.4

  ST OK. dim: 384
- **Parsers:** pdfminer/pymupdf + table extraction (camelot/tabula) + a layout-retaining schema.

Bottom line

- **Yes**, one **generic RAG/embeddings library** can serve **CVs, reports, and tenders** if you separate *document adapters* from a shared *semantic core* (retrieval + re-rank + annotation + scoring).
- Start **hybrid (dense+sparse) + cross-encoder**, add **domain-tuning** and **NLI checks**, and design from day one for **traceability** (provenance spans, scores, reasons).
- This sets you up cleanly for step-2 **form pre-fill** with auditable mappings..

---

2 | Local MVP stack (FR/EN tenders, CVs, reports)

2.1 | Retrieval (dense)

- **Multilingual small (fits easily):**
    - `intfloat/multilingual-e5-small` (~33M) or `sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2` (~118M).
- **English-optimized (if you need max quality in EN):**
    - `thenlper/gte-small` or `Alibaba-NLP/gte-base-en-v1.5` (base is fine on CPU/GPU).
- Tip: start with **`multilingual-e5-small`** for FR/EN, upgrade to `multilingual-e5-base` when you want a tiny quality boost.

2.2 | Re-ranking (cross-encoder)

- **Light & accurate:** `cross-encoder/ms-marco-MiniLM-L-6-v2` (EN).
- **Multilingual option:** `jinaai/jina-reranker-v1-base-multilingual` (base size, still comfy on 8 GB). Use it only on top-100 dense hits → top-20 final.

2.3 | Sparse retrieval (for jargon & exact clauses)

- **BM25:** `rank_bm25` (pure Python) to start.
- Later: Elastic (OpenSearch) or SPLADE if recall needs help.

2.4 | Vector store

- **FAISS** for embedded mode (simple and fast).
- Optional server mode later: **Qdrant** (Docker) when you need multi-user + filters.

2.5 | Parsers & chunking

- **PyMuPDF (`fitz`)** + metadata/page anchors.
- **Camelot**/`tabula` for tables → convert to key-value triples with cell coordinates.
- Chunk by sections/bullets; keep **(doc_id, section_path, page, bbox)** in metadata for traceability.

2.6 | Local NLI/extraction (for “does this clause match?” and pre-fill)**

- With **Ollama**: `mistral:7b-instruct` or `llama3:8b-instruct` in **Q4_K_M** quant runs on 8 GB.
- Use for: entitlement checks, short rationales, and field extraction with **constrained prompts**.

---

3 | Minimal pipeline (drop-in code)

```
# pip install sentence-transformers faiss-cpu rank-bm25 pypdf
pymupdf tqdm
from sentence_transformers import SentenceTransformer
import faiss, numpy as np
from rank_bm25 import BM25Okapi
import fitz  # PyMuPDF

# 1) Parse & chunk
def parse_pdf(path):
    doc = fitz.open(path)
    chunks = []
    for pno in range(len(doc)):
        page = doc[pno]
        text = page.get_text("blocks")  # retains block order
        for i, (_, _, _, _, t, _, _) in enumerate(text):
            t = (t or "").strip()
            if len(t) > 40:
                chunks.append({"text": t, "page": pno+1,
"block": i})
    return chunks

chunks = parse_pdf("tender.pdf")
texts = [c["text"] for c in chunks]

# 2) Dense embeddings
model = SentenceTransformer("intfloat/multilingual-e5-small")
# e5 expects "query: ..." vs "passage: ..." prefixes for best
results
passages = [f"passage: {t}" for t in texts]
E = np.vstack(model.encode(passages,
normalize_embeddings=True))

# 3) FAISS index
index = faiss.IndexFlatIP(E.shape[1])
index.add(E.astype("float32"))

# 4) BM25
bm25 = BM25Okapi([t.split() for t in texts])

# 5) Hybrid search
def hybrid_search(q, k_dense=100, k=20, alpha=0.6):
    q_dense = model.encode([f"query: {q}"],
normalize_embeddings=True)
    D, I = index.search(q_dense.astype("float32"), k_dense)
    dense_scores = {i: float(s) for i, s in zip(I[0], D[0])}
    # BM25 scores
    bm = bm25.get_scores(q.split())
    # Normalize BM25
    bm = (bm - bm.min()) / (bm.ptp() + 1e-9)
    # Fuse
    fused = []
    for i, ds in dense_scores.items():
        fs = alpha*ds + (1-alpha)*float(bm[i])
        fused.append((i, fs))
```

```
    fused.sort(key=lambda x: x[1], reverse=True)
    return [chunks[i] | {"score": s} for i, s in fused[:k]]

hits = hybrid_search("ISO 27001, MLOps platform avec MLflow et K8s")
for h in hits[:5]:
    print(h["score"], h["page"], h["text"][:120], "…")
```

> *Swap in a cross-encoder re-ranker later (e.g., `jinaai/jina-reranker-v1-base-multilingual`) on the `hits[:100]` to boost precision@5.*

---

4 | Using Ollama locally (NLI/extraction)

```
# examples: mistral & llama3 in 4-bit quant
ollama pull mistral:latest
ollama pull llama3:8b

# Python: pip install ollama
import ollama

PROMPT = """You are a compliance checker.
Clause: "{clause}"
Requirement: "Provider must be ISO 27001 certified"
Answer with JSON: {{"label":"Yes/No/Partial","rationale":
"..."}}
"""

def nli_check(clause):
    r = ollama.chat(model="mistral", messages=
[{"role":"user","content":PROMPT.format(clause=clause)}])
    return r["message"]["content"]
```

Example of response after NLI (*natural language extraction*):

```
NLI result: {
"label": "No",
"rationale": "The clause does not mention the provider's ISO
27001 certification. Therefore, it cannot be confirmed that the
provider is certified." }
```

---

5 | What fits in 8 GB VRAM (comfortably)

- **Embeddings:** “small/base” sentence-transformers (CPU or GPU).
- **Re-rankers:** MiniLM-class and multilingual base rerankers (GPU helps; CPU is fine).
- **LLM for reasoning/extraction: 7B–8B** quantized via Ollama (Q4_) — good for short answers and NLI.
- You don’t need bigger models for step-1 retrieval/ranking.

---

6 | Can the *same* lib read CVs, reports, tenders? Yes — via adapters

Keep a shared semantic core and add thin adapters:

- `TenderAdapter`: numbered requirements (MUST/SHALL), lots/eligibility, deadlines.
- `CVAdapter`: roles, durations, skills, certs; normalize to a **capability ontology** (e.g., `devops.k8s`, `mlops.mlflow`, `security.iso27001`).
- `ReportAdapter`: sections, methods, results, conclusions, annexes/tables.

All three reuse **the same**: parser → chunker → embeddings → FAISS/BM25 → (optional) reranker → scorers.

---

7 | Folder scaffold (ready to `uv/pip`)}

```
RAGGAE/
  core/
    embeddings.py          # providers (E5, GTE, bge…)
    retriever.py           # hybrid retrieve
    reranker.py            # optional cross-encoder
    index_faiss.py         # vector index
    scoring.py             # signals + weighted fit score
    nli_ollama.py          # local NLI/extractor
  io/
    pdf.py                 # PyMuPDF parsing
    tables.py              # camelot/tabula wrappers
  adapters/
    tenders.py             # parse/chunk/fields
    cv.py                  # parse/chunk/fields
    reports.py
  cli/
    index_doc.py           # index PDFs
    search.py              # query + show provenance
    quickscore.py          # tender fit score
  data/
    ontology.yaml          # skills, certs, standards
    labels/                # few-shot seeds for classifiers
```

---

8 | Early-phase eval (so you can show value next week)

- **Retrieval:** Recall@50 on 10–20 real tender questions (FR/EN).
- **Top-k quality:** nDCG@10 with cross-encoder on/off (demo the delta).
- **Classification:** Zero-shot labels (DevOps/MLOps/Lot) → quick F1 from a tiny hand-labeled set.
- **Traceability:** Every hit printed with `(doc, page, block, score)` — reviewers love this.

TL;DR

- Absolutely feasible locally: **E5-small + BM25 + optional cross-encoder**, FAISS index, **Ollama (7–8B Q4)** for NLI/extraction.
- One generic library with **adapters** lets you handle **tenders, CVs, and reports** with the same semantic core.
- Start with the code above; one can add the cross-encoder and a simple **fit score** next (must-haves met, tech match, risk flags).

---

9 | Python environment

9.1 | Check CUDA version (within `conda env torch_env`)

```
python - <<'PY'
import torch
print("PyTorch version:", torch.__version__)
print("CUDA available:", torch.cuda.is_available())
if torch.cuda.is_available():
    print("CUDA version (runtime):", torch.version.cuda)
    print("GPU:", torch.cuda.get_device_name(0))
    print("Capability:", torch.cuda.get_device_capability(0))
PY
nvidia-smi
```

The output of LX-Olivier2023:

```
PyTorch version: 2.5.1
CUDA available: True
CUDA version (runtime): 12.1
GPU: NVIDIA RTX A2000 8GB Laptop GPU
Capability: (8, 6)

+---------------------------------------------------------------
---------------------------+
| NVIDIA-SMI 580.95.05              Driver Version: 580.95.05
   CUDA Version: 13.0      |
+-----------------------------------------+---------------------
----+----------------------+
| GPU  Name                 Persistence-M | Bus-Id
Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-
Usage | GPU-Util  Compute M. |
|                                         |
   |               MIG M. |
|=========================================+=====================
====+======================|
|   0  NVIDIA RTX A2000 8GB Lap...    Off |   00000000:01:00.0
Off |                  N/A |
| N/A   49C    P8              4W /   35W |      114MiB /
8192MiB |      0%      Default |
|                                         |
   |                  N/A |
+-----------------------------------------+---------------------
----+----------------------+
```

```
+-----------------------------------------------------------------
----------------------------+
| Processes:
                            |
|  GPU   GI   CI              PID   Type   Process name
             GPU Memory |
|        ID   ID
              Usage      |
|=================================================================
============================|
|    0   N/A  N/A            6701      G   /usr/lib/xorg/Xorg
                     4MiB |
|    0   N/A  N/A            7200    C+G   ...c/gnome-remote-
desktop-daemon          83MiB |
+-----------------------------------------------------------------
----------------------------+
```

9.2 | Environment `env-adservio-raggae`

```yaml
name: adservio-raggae
channels:
  - pytorch
  - nvidia
  - conda-forge
dependencies:
  - python=3.12
  - spyder
  # Core ML stack
  - pytorch>=2.4
  - pytorch-cuda=12.1  # uses LX-Olivier2023 NVIDIA GPU (CUDA
12.x)
  - torchvision
  - torchaudio
  # RAG / retrieval
  - faiss-cpu            # simple & stable; upgrade to faiss-
gpu later if needed
  - sentence-transformers
  - numpy
  - scipy
  - scikit-learn
  - tqdm
  # PDF / parsing
  - pymupdf              # (import as `fitz`)
  - pypdf
  # utils
  - uvicorn
  - rich
  - pip
  - pip:
      - rank-bm25
      - ollama           # python client for your local Ollama
```

Use:

```
mamba env create -f env-adservio-raggae.yml
conda activate adservio-raggae
```

9.3 | Smoke test | part 1

> *The smoke test check the setup. If this runs fine, your core loop (parse → embed → index → hybrid search → provenance) is ready for plugging into adapters (tenders/CVs/reports).*

- GPU + CUDA info printed.
- Embedding shape (N, 384) and timing.
- FAISS indexed count.
- A ranked list of top matches with scores and (optional) PDF page/block.
- part 2 will show a short JSON-like verdict from the Ollama block if you enable it.

```
# -*- coding: utf-8 -*-
"""
Smoke test for RAGGAE (use )

Adservio | 2025-10-27
"""

#%% 0) Environment check (GPU, versions)
import torch, sys, platform, time
print("Python:", platform.python_version(), "| Torch:",
torch.__version__)
print("CUDA available:", torch.cuda.is_available(), "| Torch
CUDA runtime:", torch.version.cuda)
if torch.cuda.is_available():
    print("GPU:", torch.cuda.get_device_name(0), "| Compute:",
torch.cuda.get_device_capability(0))

#%% 1) Imports
from sentence_transformers import SentenceTransformer
import numpy as np, faiss
from rank_bm25 import BM25Okapi

# Optional PDF parsing
PDF_PATH = ""  # set to a local tender PDF path, e.g.,
"/home/olivi/Documents/tender.pdf"
try:
    import fitz  # PyMuPDF
except Exception as e:
    fitz = None
    print("PyMuPDF not available:", e)

#%% 2) Tiny corpus + optional PDF chunks
def parse_pdf_blocks(path, min_chars=40, max_blocks=300):
    """Return list[{'text','page','block'}] from a PDF, keeping
simple text blocks."""
    out = []
    doc = fitz.open(path)
    for pno in range(len(doc)):
        page = doc[pno]
        for bi, blk in enumerate(page.get_text("blocks")):
```

```
            # blk: (x0, y0, x1, y1, text, block_no, block_type)
            txt = (blk[4] or "").strip()
            if len(txt) >= min_chars:
                out.append({"text": txt, "page": pno+1,
"block": bi})
                if len(out) >= max_blocks:
                    return out
    return out

seed_chunks = [
    {"text": "Adservio propose une offre MLOps fondée sur
MLflow et Kubernetes (K8s).", "page": 0, "block": 0},
    {"text": "Exigence ISO 27001 et hébergement des données en
Union Européenne.", "page": 0, "block": 1},
    {"text": "DevOps CI/CD avec GitLab, ArgoCD, Helm et GitOps
pour déploiement cloud.", "page": 0, "block": 2},
    {"text": "SLA attendu 99.9%, RPO 15 minutes, RTO 1 heure.
Support 24/7 requis.", "page": 0, "block": 3},
]

if PDF_PATH and fitz:
    try:
        pdf_chunks = parse_pdf_blocks(PDF_PATH)
        print(f"Parsed {len(pdf_chunks)} blocks from PDF")
        chunks = pdf_chunks or seed_chunks
    except Exception as e:
        print("PDF parse failed, using seed chunks:", e)
        chunks = seed_chunks
else:
    if PDF_PATH and not fitz:
        print("PyMuPDF missing; set PDF_PATH='' or install it
in the env.")
    chunks = seed_chunks

texts = [c["text"] for c in chunks]
print(f"Corpus size: {len(texts)}")

#%% 3) Load embedding model (GPU if available)
MODEL = "intfloat/multilingual-e5-small"  # FR/EN good starter
device = "cuda" if torch.cuda.is_available() else "cpu"
model = SentenceTransformer(MODEL, device=device)
print("Embedding model loaded on:", device)

#%% 4) Build embeddings (timed)
t0 = time.time()
with torch.inference_mode():
    passages = [f"passage: {t}" for t in texts]  # E5-style
prefix
    E = model.encode(passages, normalize_embeddings=True,
convert_to_numpy=True, batch_size=64, show_progress_bar=False)
print("Emb shape:", E.shape, "| secs:", round(time.time()-t0,
2))

#%% 5) FAISS index (inner product / cosine with normalized
vecs)
index = faiss.IndexFlatIP(E.shape[1])
```

```
index.add(E.astype("float32"))
print("FAISS indexed:", index.ntotal, "vectors")

#%% 6) BM25 on same corpus
bm25 = BM25Okapi([t.split() for t in texts])

def _minmax(x):
    x = np.asarray(x, dtype=np.float32)
    return (x - x.min()) / (x.ptp() + 1e-9)

#%% 7) Hybrid search
def hybrid_search(query, k_dense=100, k=10, alpha=0.6):
    # dense
    qv = model.encode([f"query: {query}"],
normalize_embeddings=True,
convert_to_numpy=True).astype("float32")
    D, I = index.search(qv, min(k_dense, len(texts)))
    dense_scores = {int(i): float(s) for i, s in zip(I[0],
D[0])}
    # bm25
    bm = bm25.get_scores(query.split())
    bm = _minmax(bm)   # normalize to [0,1]
    # fuse
    fused = []
    for i, ds in dense_scores.items():
        fs = alpha*ds + (1-alpha)*float(bm[i])
        fused.append((i, fs))
    fused.sort(key=lambda x: x[1], reverse=True)
    out = []
    for i, s in fused[:k]:
        out.append(chunks[i] | {"score": round(s, 4)})
    return out

#%% 8) Run a query
query = "Plateforme MLOps avec MLflow sur Kubernetes, exigences
ISO 27001 et GitOps"
hits = hybrid_search(query, k=5)
print("\nQuery:", query)
for h in hits:
    loc = f"(p.{h['page']}, block {h['block']})" if
h.get("page") else ""
    print(f"- score={h['score']:.4f} {loc} :: {h['text']
[:110]}…")

#%% 9) Optional: quick provenance pretty-print
def show_hit(h, max_len=400):
    print(f"\n[score={h['score']}] page={h.get('page','?')}
block={h.get('block','?')}")
    print(h["text"][:max_len] + ("…" if len(h["text"])>max_len
else ""))

if hits:
    show_hit(hits[0])
```

You should read:

```
Projects/raggae/smoke_test.py
modules.json: 100%|██████████| 387/387 [00:00<00:00, 1.24MB/s]
README.md: 498kB [00:00, 63.3MB/s]
sentence_bert_config.json: 100%|██████████| 57.0/57.0
[00:00<00:00, 90.9kB/s]
config.json: 100%|██████████| 655/655 [00:00<00:00, 2.17MB/s]
model.safetensors: 100%|██████████| 471M/471M [00:13<00:00,
34.8MB/s]
tokenizer_config.json: 100%|██████████| 443/443 [00:00<00:00,
1.54MB/s]
sentencepiece.bpe.model: 100%|██████████| 5.07M/5.07M
[00:00<00:00, 10.3MB/s]
tokenizer.json: 100%|██████████| 17.1M/17.1M [00:00<00:00,
38.9MB/s]
special_tokens_map.json: 100%|██████████| 167/167 [00:00<00:00,
483kB/s]
config.json: 100%|██████████| 200/200 [00:00<00:00, 591kB/s]
Embedding model loaded on: cpu
```

9.4 | Smoke test | part 2

```python
#%% 10a) (Optional) NLI/compliance check with Ollama (requires
daemon running)
# Uncomment to test. Example: does the clause satisfy an ISO
27001 requirement?
import ollama, json
clause = hits[0]["text"] if hits else "Le prestataire dispose
d'une certification ISO 27001."
prompt = f'''You are a compliance checker.
Clause: "{clause}"
Requirement: "Provider must be ISO 27001 certified."
Answer JSON with keys: label in ["Yes","No","Partial"],
rationale (short).
'''
res = ollama.chat(model="mistral", messages=
[{"role":"user","content":prompt}])
print("\nNLI result:", res["message"]["content"])

# %% 10b) More sophisticated RAG: Hardened NLI helper
(deterministic + JSON-safe + FR/EN)
import ollama, json, re

NLI_SYS = (
  "You are a strict compliance checker. "
  "Return ONLY compact JSON with keys: label, rationale. "
  "label ∈ ['Yes','No','Partial']."
)

def parse_json_loose(s: str):
    # strip code fences and grab the first {...}
    s = s.strip()
    s = re.sub(r"^```(?:json)?\s*|\s*```$", "", s,
flags=re.I|re.M)
    m = re.search(r"\{.*\}", s, flags=re.S)
```

```
    if not m: return None
    try: return json.loads(m.group(0))
    except Exception: return None

def nli_check(clause: str, requirement: str, lang="auto"):
    prompt = (
      f"Language: {lang}. "
      f'Clause: "{clause}"\n'
      f'Requirement: "{requirement}"\n'
      'Respond as JSON:
{"label":"Yes|No|Partial","rationale":"..."}'
    )
    r = ollama.chat(
        model="mistral",
        options={"temperature": 0, "num_ctx": 4096},
        messages=[{"role":"system","content":NLI_SYS},
                    {"role":"user","content":prompt}]
    )
    out = parse_json_loose(r["message"]["content"]) or
{"label":"No","rationale":"Invalid or non-JSON output"}
    # normalize label
    lbl = out.get("label","").strip().title()
    if lbl not in {"Yes","No","Partial"}: lbl = "No"
    out["label"] = lbl
    return out

# quick test (use your best clause from hits)
clause = hits[0]["text"] if hits else "Le prestataire dispose
d'une certification ISO 27001."
print(nli_check(clause, "Provider must be ISO 27001
certified"))

# %% 10c) Batch matrix (requirements x top-k clauses)
import pandas as pd

REQUIREMENTS = [
  "Provider must be ISO 27001 certified",
  "Platform uses MLflow for MLOps",
  "Deployments on Kubernetes with GitOps",
  "Data hosted in the European Union"
]

def requirement_matrix(hits, requirements=REQUIREMENTS,
topk=5):
    rows = []
    for req in requirements:
        for i, h in enumerate(hits[:topk]):
            res = nli_check(h["text"], req)
            rows.append({
                "requirement": req,
                "hit_rank": i+1,
                "label": res["label"],
                "rationale": res["rationale"],
                "page": h.get("page"),
                "block": h.get("block"),
                "snippet": h["text"][:160].replace("\n"," ")
```

```python
            })
    df = pd.DataFrame(rows)
    # simple per-requirement verdict: first Yes > Partial > No
    order = {"Yes":2, "Partial":1, "No":0}
    verdict = (df.assign(score=df["label"].map(order))
                 .groupby("requirement")["score"].max()
                 .map({2:"Yes",1:"Partial",0:"No"}))
    return df, verdict

df_checks, verdict = requirement_matrix(hits, REQUIREMENTS,
topk=5)
print("\nOverall verdict per requirement:\n", verdict)
print("\nSample rows:\n", df_checks.head(6))


# %% 10d) Fit score from NLI labels (0..100)
label_w = {"Yes": 1.0, "Partial": 0.5, "No": 0.0}
fit_score = round(100 *
verdict.map({"Yes":1.0,"Partial":0.5,"No":0.0}).mean(), 1)
print(f"\nTender fit score (NLI): {fit_score}/100")
```

If it works well, you should read:

```
NLI result:  {
  "label": "No",
  "rationale": "The clause does not mention the provider's ISO
27001 certification. Therefore, it cannot be confirmed that the
provider is certified."
}

{'label': 'Partial', 'rationale': 'The text does not explicitly
state that Adservio is ISO 27001 certified. However, mentioning
Kubernetes (K8s) implies a certain level of compliance as it is
often used in enterprise environments where such certifications
are required.'}

Overall verdict per requirement:
 requirement
Data hosted in the European Union        Yes
Deployments on Kubernetes with GitOps    Yes
Platform uses MLflow for MLOps           Yes
Provider must be ISO 27001 certified     Yes
Name: score, dtype: object

Sample rows:
                             requirement  ...
                        snippet
0  Provider must be ISO 27001 certified  ...  Adservio propose
une offre MLOps fondée sur ML...
1  Provider must be ISO 27001 certified  ...  Exigence ISO
27001 et hébergement des données ...
2  Provider must be ISO 27001 certified  ...  DevOps CI/CD avec
GitLab, ArgoCD, Helm et GitO...
3  Provider must be ISO 27001 certified  ...  SLA attendu
99.9%, RPO 15 minutes, RTO 1 heure...
```

```
4         Platform uses MLflow for MLOps  ...  Adservio propose
une offre MLOps fondée sur ML...
5         Platform uses MLflow for MLOps  ...  Exigence ISO
27001 et hébergement des données ...

[6 rows x 7 columns]

Tender fit score (NLI): 100.0/100
```

9.5 | Troubleshooting Pytorch without CUDA

If your Spyder is still using a **CPU-only** PyTorch wheel (note `torch.version.cuda: None`). Let's fix it cleanly by installing the CUDA build from the **pytorch + nvidia** channels and avoiding any pip/conda-forge Torch that might override it.

```python
# checking torch with CUDA
import torch
print("CUDA available:", torch.cuda.is_available())
print("torch.version.cuda:", torch.version.cuda)
if torch.cuda.is_available():
    print("GPU:", torch.cuda.get_device_name(0))

# Example of misconfiguration
# ----------------------------
# Conda env : adservio-raggae
# Torch ver : 2.7.1
# torch.version.cuda : None
# CUDA available : False
# Built with CUDA? : False
# cuDNN version : None
# CUDA_VISIBLE_DEVICES: None

# additional check
import subprocess
print(subprocess.check_output(["nvidia-smi"]).decode()[:300])
# You should get something simular to:
# == nvidia-smi == nvidia-smi not callable here: Command
'['nvidia-smi', '--query-gpu=name,driver_version,cuda_version',
'--format=csv,noheader']' returned non-zero exit status 2.
```

Solution: reinstall `pytorch`

```bash
# 0) Close Spyder
# 1) Purge any CPU Torch left-overs in this env
conda activate adservio-raggae
# remove conda packages
mamba remove -y pytorch torchvision torchaudio cpuonly
# remove any pip wheels that might shadow conda packages
python -m pip uninstall -y torch torchvision torchaudio
# 2) Enforce channel priority (important)
conda config --env --set channel_priority strict
# 3) Install the CUDA build (match your runtime: 12.1)
```

```
# Do **not** add `-c conda-forge` to this command; it can pull
a CPU build.
mamba install -y -c pytorch -c nvidia \
  pytorch=2.5.* pytorch-cuda=12.1 torchvision torchaudio
# 4) (Optional) ensure Spyder can attach to this env
mamba install -y spyder-kernels
# 5) Launch Spyder from this env
```

Retest

```
# 6) Re-check in the same Spyder console
import torch, sys
print("Python exe:", sys.executable)
print("Torch:", torch.__version__)
print("torch.version.cuda:", torch.version.cuda)
print("CUDA available:", torch.cuda.is_available())
if torch.cuda.is_available():
    print("GPU:", torch.cuda.get_device_name(0))

# You should now see:
# Python exe: /home/olivi/anaconda3/envs/adservio-
raggae/bin/python3.12
# Torch: 2.5.1
# torch.version.cuda: 12.1
# CUDA available: True
# GPU: NVIDIA RTX A2000 8GB Laptop GPUVIDIA RTX A2000 8GB
Laptop GPU

```

Then point Spyder to this interpreter (same path pattern as
above).

Note on `nvidia-smi`

Inside Spyder, just check the plain command if you're curious:

```python
import subprocess, shutil
print("nvidia-smi path:", shutil.which("nvidia-smi"))
print(subprocess.check_output(["nvidia-
smi"]).decode().splitlines()[0])
```

But **PyTorch CUDA working** is the real goal; `nvidia-smi`
availability inside the IDE is optional.

This sequence fixes ~99% of "CUDA False in Spyder" cases (wrong
channel, CPU wheel, or kernel mismatch).
```
```

Just in case, a prior pin drags a CPU build back in. Make a fresh env that only uses the correct channels:

```
mamba create -n adservio-raggae12 -y -c pytorch -c nvidia -c conda-forge \
  python=3.12 pytorch=2.5.* pytorch-cuda=12.1 torchvision torchaudio \
  faiss-cpu sentence-transformers numpy scipy scikit-learn tqdm \
  pymupdf pypdf uvicorn rich spyder-kernels

mamba activate adservio-raggae12
spyder &
```

9.6 | Troubleshooting numpy `broadcast_to` import error.`

This error means your NumPy install in `adservio-raggae` is inconsistent (CPU-only Torch is fine now). Let's fix NumPy cleanly and avoid mixed pip/conda wheels.

Quick check (run in the same Spyder console)

```python
import numpy, sys
print("NumPy:", numpy.__version__, "| path:", numpy.__file__)
```

Fix (close Spyder, then in a terminal)

```
mamba activate adservio-raggae

# 1) Remove any pip wheel that might be shadowing conda's NumPy
python -m pip uninstall -y numpy

# 2) Install a clean conda-forge build compatible with Python 3.12
# (both 1.26.x and 2.1.x work; 2.1.x is current and stable)
mamba install -y -c conda-forge "numpy=2.1.*" "scipy>=1.11"

# Optional: harmonize key libs to conda-forge to avoid ABI mismatches
mamba install -y -c conda-forge scikit-learn tqdm pymupdf pypdf faiss-cpu sentence-transformers
```

> *If you prefer staying on NumPy 1.x: use `numpy=1.26.*` (it supports Python 3.12). `broadcast_to` exists in both.*

Relaunch & re-test

1. Start Spyder **from the env**:

```
spyder &
```

1. In Spyder, run:

```python
import numpy as np
from sentence_transformers import SentenceTransformer
print("NumPy OK:", np.__version__)
m = SentenceTransformer("intfloat/multilingual-e5-small")
print("ST OK. dim:", m.get_sentence_embedding_dimension())

# you should see
# NumPy OK: 1.26.4
# ST OK. dim: 384
```

If it still fails, show me the output of:

```
mamba list | egrep 'numpy|scipy|torch|sentence|faiss'
```

and we'll zero in—but in 99% cases the clean conda-forge NumPy reinstall resolves the `broadcast_to` import error.