GATE Prep RAG (LangChain)

An end-to-end Retrieval-Augmented Generation (RAG) pipeline for **GATE DA/AI&ML** preparation.

- Asks you for a **topic** (from the **Portions** page) and a **per-question time limit**.
- Generates MCQs from the syllabus context, asks the questions, times your answers, and **explains mistakes + fixes them**.

1) Environment Setup

We install only lightweight dependencies. By default we use **HuggingFace local** models

```
!pip -q install langchain langchain-community langchain-text-splitters
chromadb pypdf sentence-transformers scikit-learn rank-bm25
inputimeout transformers accelerate torch --upgrade
# Optional (only if you want to use Groq as a fallback):
!pip -q install langchain-groq
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 0.0/67.3 kB ? eta -:--:--
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 67.3/67.3 kB 2.8 MB/s eta
0:00:00
ents to build wheel ... etadata (pyproject.toml) ...
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.5/2.5 MB 48.2 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 19.8/19.8 MB 56.9 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 310.5/310.5 kB 22.1 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 9.5/9.5 MB 65.9 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 284.2/284.2 kB 14.3 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.9/1.9 MB 59.7 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 103.3/103.3 kB 8.2 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 16.5/16.5 MB 66.5 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 72.5/72.5 kB 5.3 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 105.4/105.4 kB 7.7 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 71.6/71.6 kB 5.9 MB/s eta
0:00:00
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 64.7/64.7 kB 4.6 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 510.8/510.8 kB 25.9 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 50.9/50.9 kB 3.4 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 4.7/4.7 MB 75.2 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 452.2/452.2 kB 22.0 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 46.0/46.0 kB 2.7 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 86.8/86.8 kB 5.9 MB/s eta
0:00:00
l) ... ERROR: pip's dependency resolver does not currently take into
account all the packages that are installed. This behaviour is the
source of the following dependency conflicts.
google-colab 1.0.0 requires requests==2.32.4, but you have requests
2.32.5 which is incompatible.
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.9/134.9 kB 5.3 MB/s eta
0:00:00
```

#2) Config — Choose LLM & Paths

- **LLM_PROVIDER** can be `'hf'` (HuggingFace local, default), `'ollama'`, or `'groq'`.
- **Syllabus PDF**: we use your uploaded *Portions* file.

```
import os, sys, json, time, textwrap, re, math, random, uuid
from pathlib import Path

LLM_PROVIDER = os.environ.get("LLM_PROVIDER", "hf")  # 'hf' (default),
'ollama', or 'groq'
SYLLABUS_PDF = "/content/GATE _DA_2025_Syllabus (1).pdf"  # <--
Portions page you uploaded
VECTOR_DIR = "./chroma_gate_portions"

# We *read* your existing env keys if present. No need to change how
you set them.
OPENAI_API_KEY   = os.environ.get("OPENAI_API_KEY")
GEMINI_API_KEY   = os.environ.get("GEMINI_API_KEY")
GROQ_API_KEY     = os.environ.get("GROQ_API_KEY")
TOGETHER_API_KEY = os.environ.get("TOGETHER_API_KEY")
```

3) LLM Initialization (Local-first, token-light)

We keep it **cheap**:

- **HuggingFace local** (default): uses `google/flan-t5-base` — small, runs locally, no tokens.

- **Ollama** (if installed): set `LLM_PROVIDER='ollama'` and have a local model pulled (e.g., `ollama pull llama3.2:3b`).
- **Groq** (optional): set `LLM_PROVIDER='groq'` and export `GROQ_API_KEY`. Uses `llama-3.1-8b-instant`.

```
from langchain_community.llms import HuggingFacePipeline
from langchain_community.chat_models import ChatOllama
from langchain_groq import ChatGroq
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM,
pipeline

def get_llm(provider=LLM_PROVIDER):
    if provider == "ollama":
        try:
            return ChatOllama(model=os.environ.get("OLLAMA_MODEL",
"llama3.2:3b"), temperature=0.2)
        except Exception as e:
            print("Ollama not available or model missing. using to HF
local.")
    if provider == "groq" and (os.environ.get("GROQ_API_KEY")):
        return ChatGroq(model="llama-3.1-8b-instant", temperature=0.2)
    # Default: HuggingFace local (no API keys needed)
    model_name = os.environ.get("HF_MODEL", "google/flan-t5-base")
    tok = AutoTokenizer.from_pretrained(model_name)
    mdl = AutoModelForSeq2SeqLM.from_pretrained(model_name)
    gen = pipeline("text2text-generation", model=mdl, tokenizer=tok,
max_new_tokens=256)
    return HuggingFacePipeline(pipeline=gen)

llm = get_llm()
print("Using LLM:", type(llm).__name__)
```

```
Device set to use cpu
```

```
Using LLM: HuggingFacePipeline
```

4) **Indexing**

1. **Chunk Optimization (Semantic Splitter)** — split the PDF into semantically coherent chunks.

2. *Multi-representation Indexing (Parent Document, Dense X)** — we keep both raw text + a compact summary for each chunk.

3. **Specialized Embeddings** — use **`sentence-transformers/all-MiniLM-L6-v2`** (free + local).

4. **Hierarchical Indexing (RAPTOR-like)** — build a *summary tree* (coarse → fine) so retrieval can work at multiple levels.

```
from pypdf import PdfReader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from sentence_transformers import SentenceTransformer
from sklearn.cluster import KMeans
import numpy as np
import chromadb
from chromadb.config import Settings


def load_pdf_text(path):
    reader = PdfReader(path)
    pages = [p.extract_text() or "" for p in reader.pages]
    text = "\n".join(pages)
    return text, pages

raw_text, raw_pages = load_pdf_text(SYLLABUS_PDF)


splitter = RecursiveCharacterTextSplitter(
    chunk_size=800, chunk_overlap=120, separators=["\n\n", "\n", ". ",
"; ", ", "]
)
chunks = splitter.split_text(raw_text)

def summarize_local(text, llm_obj):
    prompt = f"Summarize in 2 short bullet points:\n{text[:1200]}"
    try:

        if hasattr(llm_obj, "invoke"):
            return llm_obj.invoke(prompt) if
isinstance(llm_obj.invoke(prompt), str) else
str(llm_obj.invoke(prompt))
        else:
            return llm_obj(prompt)
    except Exception:
        return text[:200]

chunk_summaries = [summarize_local(c, llm) for c in chunks]


embed_model = SentenceTransformer("sentence-transformers/all-MiniLM-
L6-v2")

def embed_texts(texts):
    return embed_model.encode(texts, convert_to_numpy=True,
show_progress_bar=False)

chunk_embeddings = embed_texts(chunks)

def build_hierarchy(chunks, embeddings, n_parents=6):
```

```
    n = len(chunks)
    k = min(n_parents, max(2, n//5))  # heuristic
    km = KMeans(n_clusters=k, n_init=5, random_state=42)
    labels = km.fit_predict(embeddings)
    parents = []
    for lab in range(k):
        idxs = np.where(labels==lab)[0].tolist()
        group_text = "\n".join(chunks[i] for i in idxs)[:3000]
        parent_summary = summarize_local(group_text, llm)
        parents.append({
            "label": int(lab),
            "child_ids": idxs,
            "summary": parent_summary
        })
    return parents, labels

parents, labels = build_hierarchy(chunks, chunk_embeddings)

client = chromadb.PersistentClient(path=VECTOR_DIR,
settings=Settings(anonymized_telemetry=False))
if "gate_portions" in [c.name for c in client.list_collections()]:
    client.delete_collection("gate_portions")

collection = client.create_collection("gate_portions",
metadata={"hnsw:space": "cosine"})

metas = []
for i, (txt, summ, lab) in enumerate(zip(chunks, chunk_summaries,
labels)):
    metas.append({
        "id": str(i),
        "summary": summ if isinstance(summ, str) else str(summ),
        "cluster": int(lab)
    })
collection.add(
    ids=[m["id"] for m in metas],
    embeddings=[e.tolist() for e in chunk_embeddings],
    documents=chunks,
    metadatas=metas
)


HIER_PATH = Path(VECTOR_DIR) / "parents.json"
HIER_PATH.write_text(json.dumps(parents, indent=2))
print(f"Indexed {len(chunks)} chunks. Parents saved to {HIER_PATH}.")
```

{"model_id":"1f34caca734a4277b7f29cf9e48e4f7b","version_major":2,"vers
ion_minor":0}

```
{"model_id":"2be5d74e1ecf4f22bd6238858eb49750","version_major":2,"version_minor":0}

{"model_id":"fc06729af3644e3aa03898383575573a","version_major":2,"version_minor":0}

{"model_id":"f39ee418dd4f4e009668a2563a7e5dd2","version_major":2,"version_minor":0}

{"model_id":"669656eed8134b51b5e0f03c159773c3","version_major":2,"version_minor":0}

{"model_id":"7e74d479bab445ea8672afa9a330accc","version_major":2,"version_minor":0}

{"model_id":"73584c8272df458db4d946e734993ad1","version_major":2,"version_minor":0}

{"model_id":"345895cc79a24dc4a15f416b9559fc12","version_major":2,"version_minor":0}

{"model_id":"26c503134d4b45f1a595d0e9d8402985","version_major":2,"version_minor":0}

{"model_id":"24256355604b40d988d875b55357d5f9","version_major":2,"version_minor":0}

{"model_id":"1cc392c13da4406894fff4c60cd1b22c","version_major":2,"version_minor":0}
```


```
Indexed 5 chunks. Parents saved to chroma_gate_portions/parents.json.
```

5) **Routing**

- **Logical routing** — choose among routes (Vector store vs. parent summaries).
- **Semantic routing** — choose the prompt template (MCQ generation vs. explanation) based on the intent.

```python
from rank_bm25 import BM25Okapi

# Utility: choose route based on query length / specificity
def logical_route(query:str):
    # If the query is broad, first consult parent summaries.
    if len(query.split()) < 3 or any(k in query.lower() for k in ["overview","summary","syllabus","portions"]):
        return "parents"
    return "vector"

def semantic_route(query:str):
    q = query.lower()
```

```
    if any(k in q for k in
["mcq","question","quiz","test","practice"]):
        return "mcq"
    if any(k in q for k in ["explain","why","understand","how"]):
        return "explain"
    return "mcq"
```

6) **Query Translation**

We implement:

- **Query Decomposition (Multi-query, Step-back, RAG-Fusion)** — expand the user topic into multiple focused sub-queries.

- **Pseudo-documents (HyDE)** — synthesize a hypothetical short note for better retrieval, then embed it and search.

```
from sklearn.feature_extraction.text import TfidfVectorizer

def multiquery_expand(topic:str):
    base = topic.strip()
    qs = [
        base,
        f"{base} definitions and key formulae",
        f"{base} common mistakes and pitfalls",
        f"{base} important theorems and properties",
        f"{base} examples and solved problems"
    ]
    return qs

def step_back(topic:str):
    return f"Explain the fundamental ideas behind {topic} for GATE
preparation."

def hyde_pseudo_doc(topic:str):
    prompt = f"""Write 4 concise bullet points as if they were a short
study note for: {topic}.
Use exact terms from the GATE syllabus if relevant.
"""
    try:
        if hasattr(llm, "invoke"):
            out = llm.invoke(prompt)
            return out if isinstance(out, str) else str(out)
        else:
            return llm(prompt)
    except Exception:
        return topic

def embed_query(q:str):
```

```
    return embed_texts([q])[0]

def retrieve_with_translation(topic:str, k=6):
    # Compose multi-queries + step-back + HyDE
    queries = multiquery_expand(topic) + [step_back(topic)]
    hyde_doc = hyde_pseudo_doc(topic)
    queries.append(hyde_doc)

    q_embs = embed_texts(queries)
    # Search in both parents and chunks (logical route mix)
    # 1) Vector search in chunks
    res = collection.query(query_embeddings=q_embs.tolist(),
n_results=k)
    # 2) Parent summaries with BM25 as coarse ranker
    parent_texts = [p["summary"] for p in parents]
    bm25 = BM25Okapi([t.split() for t in parent_texts])
    scores = bm25.get_scores(hyde_doc.split())
    parent_hits = np.argsort(scores)[::-1][:min(k,
len(parents))].tolist()

    # Merge results (RAG-Fusion-like)
    docs = []
    seen = set()
    for ids, docs_list in zip(res["ids"], res["documents"]):
        for cid, d in zip(ids, docs_list):
            if cid not in seen:
                seen.add(cid)
                docs.append((cid, d))
    for ph in parent_hits:
        for child in parents[ph]["child_ids"][:2]:  # take a couple
child docs from top parents
            if str(child) not in seen:
                seen.add(str(child))
                d = collection.get(ids=[str(child)])["documents"][0]
                docs.append((str(child), d))
    return docs[:k]
```

7) **Retrieval**

- **Ranking / Re-ranking** — we combine vector similarity with a lightweight BM25 + cosine scoring.

- **Refinement (CRAG-style)** — if confidence is low, we expand the search (e.g., broaden query terms).

- **Active retrieval** — if we still have low confidence, we re-run retrieval with relaxed filters.

```
from sklearn.metrics.pairwise import cosine_similarity
```

```
def score_doc(query:str, doc:str):
    emb_q = embed_query(query).reshape(1,-1)
    emb_d = embed_texts([doc]).reshape(1,-1)
    cos = cosine_similarity(emb_q, emb_d)[0][0]
    # Add BM25 lightweight term overlap bonus
    bm25 = len(set(query.lower().split()) & set(doc.lower().split()))
/ (len(query.split())+1)
    return 0.8*cos + 0.2*bm25

def refined_retrieve(topic:str, k=6, threshold=0.35):
    docs = retrieve_with_translation(topic, k=k*2)  # broader initial
pool
    scored = [(did, d, score_doc(topic, d)) for did, d in docs]
    scored.sort(key=lambda x: x[2], reverse=True)
    top = scored[:k]
    conf = np.mean([s for _,_,s in top]) if top else 0.0
    if conf < threshold:
        # CRAG-style expansion: broaden query keywords
        broader = topic + " fundamentals basics introduction
properties"
        docs2 = retrieve_with_translation(broader, k=k*2)
        scored2 = [(did, d, score_doc(broader, d)) for did, d in
docs2]
        scored2.sort(key=lambda x: x[2], reverse=True)
        top = (top + scored2[:k])
        top.sort(key=lambda x: x[2], reverse=True)
        top = top[:k]
    return top  # list of (id, doc, score)
```

8) **Generation**

- **Self-RAG** — LLM first drafts an answer; if it thinks context is insufficient, it requests **re-retrieval**.

- **RRR loop** (Reflect → Re-retrieve → Refine) — we iterate once if needed.

```
def self_rag_answer(question:str, topic:str):
    # Retrieve
    top_docs = refined_retrieve(topic, k=6)
    ctx = "\n\n".join(d for _,d,_ in top_docs)
    prompt = f"""You are a GATE tutor. Use ONLY the context to answer.
Context:
{ctx}

Question: {question}
If context is insufficient, say "NEED_MORE_CONTEXT".
"""
    if hasattr(llm, "invoke"):
        ans = llm.invoke(prompt)
```

```
    else:
        ans = llm(prompt)
    if isinstance(ans, dict):
        ans = ans.get("content", str(ans))
    if "NEED_MORE_CONTEXT" in str(ans):
        # Re-retrieve with broader context
        top_docs = refined_retrieve(topic + " overview core ideas",
k=6)
        ctx = "\n\n".join(d for _,d,_ in top_docs)
        prompt2 = f"""Context (broadened):
{ctx}

Question: {question}
Answer concisely and correctly for GATE.
"""
        ans2 = llm.invoke(prompt2) if hasattr(llm, "invoke") else
llm(prompt2)
        return str(ans2)
    return str(ans)
```

9) **Query Construction**

We support the **VectorDB self-query retriever** path (natural language → metadata filters). Here, we derive a simple **topic filter** from the syllabus to steer retrieval.

```
# Extract "topics" (coarse) from the syllabus text
def extract_topics(text):
    # Split by sections based on known headings (heuristic)
    heads = ["Probability and Statistics", "Linear Algebra", "Calculus
and Optimization",
             "Programming, Data Structures and Algorithms", "Database
Management and Warehousing",
             "Machine Learning", "AI: Search"]
    topics = {}
    for h in heads:
        pat = h.split(":")[0]
        m = re.search(pat, text, flags=re.I)
        if m:
            topics[h] = h
    # Fallback if regex misses
    if not topics:
        topics = {"General": "GATE syllabus portions"}
    return list(topics.keys())

TOPICS = extract_topics(raw_text)
print("Detected Portion Topics:", TOPICS)

def apply_topic_filter(topic, docs):
```

```
    t = topic.lower().split(":")[0]
    out = []
    for did, d, s in docs:
        if t in d.lower():
            out.append((did,d,s+0.05))
        else:
            out.append((did,d,s))
    out.sort(key=lambda x: x[2], reverse=True)
    return out
```

```
Detected Portion Topics: ['Probability and Statistics', 'Linear
Algebra', 'Calculus and Optimization', 'Programming, Data Structures
and Algorithms', 'Database Management and Warehousing', 'Machine
Learning', 'AI: Search']
```

10) MCQ Generation, Timed Quiz, and Feedback on Mistakes

- We generate MCQs from retrieved context, ask within your **time limit**, and grade your answer.

- If wrong/late, we give **what mistake you made** and the **correct reasoning/answer**.

```
from inputimeout import inputimeout, TimeoutOccurred

def build_mcq_from_context(topic:str, n=5):
    top_docs = apply_topic_filter(topic, refined_retrieve(topic, k=6))
    ctx = "\n\n".join(d for _,d,_ in top_docs)
    prompt = f"""Create {n} **multiple-choice** questions for GATE on
the topic "{topic}".
Use only this context:
{ctx}

Format strictly as JSON list with each item:
{{
  "question": "...",
  "options": ["A) ...", "B) ...", "C) ...", "D) ..."],
  "answer": "A/B/C/D",
  "explanation": "short explanation using the context",
  "common_mistake": "typical mistake and why it's wrong"
}}
Keep them precise and syllabus-aligned.
"""
    txt = llm.invoke(prompt) if hasattr(llm, "invoke") else
llm(prompt)
    if not isinstance(txt, str):
        txt = str(txt)
    # Attempt to parse JSON
```

```
    try:
        data = json.loads(re.findall(r'\[.*\]', txt, flags=re.S)[0])
        return data


def ask_quiz(topic:str, per_question_time:int=45, n_questions:int=5):
    mcqs = build_mcq_from_context(topic, n=n_questions)
    score = 0
    results = []
    print(f"\nTopic: {topic}  |  Time per question: 
{per_question_time}s")
    for i, q in enumerate(mcqs, 1):
        print(f"\nQ{i}. {q['question']}")
        for opt in q["options"]:
            print(opt)
        start = time.perf_counter()
        try:
            ans = inputimeout(prompt="Your answer (A/B/C/D): ", 
timeout=per_question_time).strip().upper()
            elapsed = time.perf_counter() - start
            timed_out = False
        except TimeoutOccurred:
            ans = None
            elapsed = per_question_time
            timed_out = True

        correct = (ans == q["answer"])
        if timed_out:
            print("□ Time up!")
            feedback = f"You ran out of time. Correct answer: 
{q['answer']}."
        elif not correct:
            feedback = f"□ Incorrect. You chose {ans}, but correct is 
{q['answer']}."
        else:
            feedback = "□ Correct!"

        # Explain mistake + correction
        if timed_out or not correct:
            feedback += f"\n**What went wrong:** 
{q.get('common_mistake','Looked away from key property.')}"
            feedback += f"\n**Fix:** {q.get('explanation','Review the 
definition/proof from context.')}"
        else:
            score += 1

        print(feedback)
        results.append({
            "question": q["question"],
            "your_answer": ans,
```

```
            "correct": q["answer"],
            "timed_out": timed_out,
            "explanation": q["explanation"]
        })
    print(f"\nScore: {score}/{len(mcqs)}")
    return results
```

11) Run the Tutor

Pick a **topic** and a **time limit**. Topics are auto-detected from the *Portions* page.

```
print("Available Portion Topics:")
for i, t in enumerate(TOPICS, 1):
    print(f"{i}. {t}")
topic_index = int(input("Choose a topic number: ").strip())
time_per_q = int(input("Time per question (seconds): ").strip() or
"45")
results = ask_quiz(TOPICS[topic_index-1],
per_question_time=time_per_q, n_questions=5)
```

```
Available Portion Topics:
1. Probability and Statistics
2. Linear Algebra
3. Calculus and Optimization
4. Programming, Data Structures and Algorithms
5. Database Management and Warehousing
6. Machine Learning
7. AI: Search
Choose a topic number: 4
Time per question (seconds): 10

Token indices sequence length is longer than the specified maximum
sequence length for this model (939 > 512). Running this sequence
through the model will result in indexing errors


Topic: Programming, Data Structures and Algorithms  |  Time per
question: 10s

Q1. Which distribution has mean=variance?
A) Binomial
B) Poisson
C) Normal
D) Uniform
Your answer (A/B/C/D): □ Incorrect. You chose , but correct is B.
**What went wrong:** Choosing Normal; Normal's variance is σ², not
tied to the mean.
**Fix:** For Poisson(λ), mean = variance = λ.
```

```
Score: 0/1
```