ChatGPT

# HackRx 6.0 Hackathon – LLM-Based Query System

**Hackathon Objective:** HackRx 6.0 is a generative-AI hackathon (by Bajaj Finserv Health) that challenges participants to build an **"LLM-Powered Intelligent Query–Retrieval System"** [1] [2]. The system must accept natural-language insurance queries (e.g. "46-year-old male, knee surgery in Pune, 3-month-old policy"), parse key details, and **retrieve relevant information from large unstructured documents** (policy wordings, contracts, emails, etc.) [1] [3]. Crucially, the system should use semantic search (not just keyword matching) and make human-like inferences: for a query it must determine an outcome (e.g. "approved" or "rejected") and **output a structured JSON** containing **Decision**, **Amount** (if any), and **Justification** (mapping each decision to the exact clause(s) used) [3] [4]. For example, if a 46M patient with knee surgery falls under a covered clause, the system should return something like `{ "Decision": "Yes", "Amount": 150000, "Justification": "Clause X covers knee replacements (see page 12)" }`. As the official problem statement notes, **explaining the decision by citing the exact clauses** is required [4].

**Key Requirements:** According to the HackRx site, the system must: - Handle **vague or incomplete English queries** (interpreting "46M knee surgery, Pune" meaningfully) [5]. - Ingest diverse document formats (PDFs, Word files, emails). - Perform **semantic retrieval** from these documents and then reason (often via an LLM) to decide coverage and outputs [1] [3]. - Produce a **consistent, interpretable JSON** for downstream use (claims processing, audit) [3] [4].

# Technology and Models (Local Setup)

Given your laptop (16 GB RAM, GTX 1650 GPU, ~4 GB VRAM), the recommended approach is a **Retrieval-Augmented Generation (RAG)** pipeline using open-source tools. In a RAG system, you first index (embed) the documents into a vector database, then at query time retrieve relevant chunks and feed them to an LLM to generate the answer [6] [7]. Popular frameworks like **LangChain** or HuggingFace's Transformers can help implement this.

- **Document Ingestion & Vector Store:** Use Python libraries (e.g. PyMuPDF or pdfplumber to extract text from PDFs/Word). Split large documents into smaller passages (LangChain's TextSplitters) and embed each chunk with a sentence-transformer (e.g. `all-MiniLM-L6-v2`) to get vector embeddings [6]. Store these vectors in a local vector database (FAISS or similar). This lets you perform semantic search over the policy text.

- **Model for Question Answering:** For the LLM, pick an open model that can run on 4 GB VRAM. Empirical guidance suggests staying at or below the ~7B parameter range and using quantization if needed [8]. For example, **Mistral 7B** (with 4-bit quantization) can run on a 1650 GPU, and smaller models like **Llama-3 3B** or **Qwen-2.5B** fit comfortably [8]. Tools like **Ollama** or Hugging Face's `transformers` + `bitsandbytes` can load and quantize these models on modest GPUs [9] [8]. (In fact, recent articles note that tools like Ollama allow running models like Llama2 or Mistral locally even on a 1650 4GB GPU [9].) Choose a model that balances size and performance (for example, a 4-

bit quantized Llama-2 7B Chat model, or a 3B-5B model, both of which can fit in 4 GB VRAM with proper optimization [8] ).

- **Frameworks:** Use Python. Install libraries like `langchain`, `transformers`, `sentence-transformers`, and a vector DB (e.g. `faiss`). LangChain provides high-level chains (e.g. `RetrievalQA`) but you can also code the steps manually. The LangChain tutorial outlines exactly the RAG pipeline steps – "Load, Split, Store" for indexing, then "Retrieve, Generate" at query time [6] [7] . It also has guided code samples (see Installation step for pip commands).

- **QA and Output:** Prompt the LLM to **output JSON**. In your prompt, include instructions like: *"Given the following query and relevant policy text, output a JSON with keys Decision, Amount, and Justification. Justification should quote the exact clause text used."* LangChain's output-parsing (e.g. Structured Outputs) can enforce JSON format. (For instance, you can instruct the model to respond in strict JSON and even validate it.) This ensures your final answer is machine-readable.

# Step-by-Step Development Guide

1. **Environment Setup:** Install Python and necessary libraries. For example:
2. `pip install torch transformers sentence-transformers langchain faiss-cpu`.
3. If using GPU, install a PyTorch version with CUDA support.

4. (Optional) Tools like Ollama can simplify local LLM hosting. Ollama provides command-line or API access to pre-built LLMs and handles quantization automatically [9] .

5. **Data Collection:** Obtain the sample policy documents. (HackRx provides sample docs; you can also download relevant insurance PDFs from IRDAI or Bajaj Finserv sites.) For each PDF/Word:

6. Extract text (e.g. using `PyMuPDF`, `pdfplumber`, or `python-docx`).

7. Clean and standardize text (remove headers/footers, handle formatting).

8. **Document Splitting & Embedding:** Break each document into manageable passages (e.g. 500–1000 characters or by clause). Use LangChain's TextSplitters or similar. Then, compute embeddings for each passage with a sentence-transformer model (e.g. `sentence-transformers/all-MiniLM-L6-v2` or any compact model). These embeddings capture semantic meaning.

9. **Vector Indexing:** Store all passage embeddings in a vector index (e.g. FAISS). Tag each vector with metadata (document ID, passage text, clause number). This index allows fast semantic search: for a given query embedding, FAISS returns the most relevant passages. LangChain's `VectorStore` classes (InMemory/FAISS) automate this [6] .

10. **Query Handling:** When a user query arrives:

11. (Optional) **Parse the query.** You can either parse manually (e.g. regex to extract "46-year-old", "knee surgery", "Pune", "3-month policy") or rely on the LLM to interpret it. A structured prompt can help the LLM identify key attributes.

12. Convert the query into an embedding (using the same sentence-transformer).

13. Retrieve the top-N relevant passages from the vector index (by cosine similarity). These passages should contain relevant clauses or rules.

14. **LLM Prompting and Answer Generation:** Combine the retrieved passages into a context. Construct a prompt such as:

    > "We have the following policy clauses:\n> `[clause text 1] [clause text 2] ...` \n> *Query:* "46-year-old male, knee surgery, Pune, 3-month policy."\n> *Task:* Determine if this claim is approved, and specify the amount if approved. Provide the answer as JSON: `{ "Decision": ..., "Amount": ..., "Justification": ... }`. Include in Justification the exact clauses used."
    >
    > Pass this prompt to the LLM (e.g. Llama-2 or Mistral via Transformers or Ollama). The model should read the clauses and query, then output JSON with Decision, Amount, and quoted clause text for Justification. (Because the prompt explicitly asks for JSON, the model's response can be parsed directly.)

15. **Structured Output:** Ensure the model's response is valid JSON. You may run a quick JSON parse or use LangChain's output parsers for a schema. The output must match the hackathon requirement: a **decision ("approved"/"rejected")**, any payout amount, and clause references. For example:

    ```
    {
      "Decision": "Approved",
      "Amount": 120000,
      "Justification": "Covered under policy clause 5.2: 'Knee replacement surgery is included after 3-month waiting period.'"
    }
    ```

    By citing the clause text exactly, you meet the "explain with exact clauses" requirement [4].

16. **Testing:** Verify with sample queries. The HackRx page gives a sample query ("46M, knee surgery..." with sample answer "Yes, covered"). Make sure your system produces the expected JSON for this and other test cases. Adjust your prompts or retrieval if needed (e.g., include more context, increase passages, refine splitting).

# Running Locally on GTX 1650/16GB

Your hardware is modest but capable with the right model choices. The GTX 1650 has ~4 GB VRAM – sufficient for small LLMs. The HuggingFace forum notes that **models below ~7B parameters** can run on 4 GB (with slight VRAM shortfall), and recommends using 4-bit quantization [8]. In practice: - Use a 4-bit quantized LLM. Tools like **Ollama** can automatically download and quantize Llama-2 or Mistral models for a 4 GB GPU [9]. - Consider **Llama-2 7B Chat** (quantized to 4-bit with BitsAndBytes) or **Mistral 7B** (native quantized) or even smaller (Llama-3 3B or Qwen-2.5B). For example, Llama-2 7B in 4-bit requires ~3.6 GB RAM [8]. Alternatively, open-source chat models like **Vicuna-7B** or **Orca-Mini** might also work. - If

necessary, run inference on CPU (slower) or split work between CPU and GPU. But with careful quantization and low-batch sizes, the GPU should handle the generation in real time for single queries.

Finally, remember to manage VRAM: avoid unnecessarily large context lengths, and offload embeddings or text processing to CPU when possible.

# Summary

In summary, you should build a RAG-based QA system:

- **Index policy documents:** extract text → split into chunks → embed → store in FAISS.
- **Query pipeline:** embed query → retrieve relevant chunks → prompt LLM with clauses + query.
- **Answer generation:** LLM produces **JSON** with Decision, Amount, Justification (citing clause text).
- **Local model use:** Select a small open LLM (4–7B) with quantization. Tools like Ollama/LangChain enable running these on a GTX1650 [9] [8].
- **Programming:** Use Python, HuggingFace Transformers or LangChain for implementation. For example, LangChain's RetrievalQA chain automatically wires embedding, vector DB, and LLM together in code [6] [7].

By following these steps and focusing on semantic search + reasoning, you'll meet the HackRx requirements. The final deliverable is a working prototype that takes an English query, accesses local insurance documents, and outputs a precise JSON decision with clause-backed justification [3] [4].

**Sources:** Official HackRx 6.0 problem description and guidelines [1] [3]; LangChain RAG tutorial (retrieval+generation steps) [6] [7]; examples of running local LLMs on GTX1650 [9] [8].

---


[1] HackRx 6.0 Hackathon by Bajaj Finserv – National-Level Hackathon for Engineering Students | Talentd
https://www.talentd.in/articles/hackrx-6-0-hackathon-by-bajaj-finserv-national-level-hackathon-for-engineering-students

[2] [3] [4] [5] HackRx 6
https://hackrx.in/

[6] [7] Build a Retrieval Augmented Generation (RAG) App: Part 1 | LangChain
https://python.langchain.com/docs/tutorials/rag/

[8] Best LLMs that can run on 4gb VRAM - Beginners - Hugging Face Forums
https://discuss.huggingface.co/t/best-llms-that-can-run-on-4gb-vram/136843

[9] Running an Large Language Model(llama2/Mistral) on Your Laptop with GTX 1650 and Ollama | by Pranay Waghmare | Medium
https://medium.com/@pranay1001090/running-an-large-language-model-llama2-mistral-on-your-laptop-with-gtx-1650-and-ollama-8b90e8aa8664