LLMExam: An LLM-Powered System for PhD Entrance Examination Questions

Alankrit Singh
`f20221186@goa.bits-pilani.ac.in`

Divyanshu Singh
`f20221129@goa.bits-pilani.ac.in`

Saumya Kathuria
`f20221106@goa.bits-pilani.ac.in`

Manan Jain
`f20220638@goa.bits-pilani.ac.in`

Kshitij Simha R
`f20220572@goa.bits-pilani.ac.in`

Sravan Danda
`dandas@goa.bits-pilani.ac.in`

Aditya Challa
`adityac@goa.bits-pilani.ac.in`

Department of Computer Science and Information Systems,
BITS Pilani, K.K. Birla Goa Campus,
Goa, India

June 29, 2025

Contents

Abstract

Faculty responsible for recruiting PhD students often need to create entrance examinations using questions from various sources, typically stored as PDF documents from past tests or question banks. Manually searching and curating questions based on specific topics, concepts, or difficulty levels from these large documents is a tedious and time-consuming process. This report presents LLMExam, a system developed to assist faculty by leveraging Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG). LLMExam automates the extraction of individual questions as images from source PDFs using Optical Character Recognition (OCR) and layout analysis. A multimodal LLM (Qwen2.5-VL-7B-Instruct) analyzes each question image to generate relevant metadata tags: Subject, Topic, Concept, and a calculated Difficulty score. These tags, combined with weighted semantic embeddings (`thenlper/gte-large`), are stored in a FAISS vector database, creating a searchable question repository. Faculty can query the system using natural language (e.g., "Find 3 hard questions on Advanced Algorithms") via a query processing module that interprets the request. The system then retrieves the most relevant question images based on semantic similarity and filters them by the specified difficulty, providing faculty with suitable questions for constructing entrance tests. This project demonstrates a practical application of multimodal LLMs and RAG for efficient question bank management and curation in an academic recruitment context.

1 Introduction

1.1 Problem Statement

Faculties involved in the PhD student recruitment process often face the challenge of assembling relevant and balanced entrance examinations. Question sources, such as previous years' papers or specialized question banks, are frequently available only as large PDF documents. Manually sifting through these documents to find questions that match specific criteria – subject area, sub-topic, underlying concept, and appropriate difficulty level – is inefficient and labor-intensive. This process hinders the timely creation of tailored tests that accurately assess the aptitude of PhD applicants in desired research domains.

1.2 Motivation

The need for a more efficient method for curating PhD entrance exam questions motivated this project. Advances in multimodal Large Language Models (LLMs) that understand both text and images, coupled with Retrieval-Augmented Generation (RAG) techniques and vector databases for semantic search, offer a technological solution. The goal was to build a tool, named LLMExam, to automate the indexing and retrieval of questions, thereby empowering faculty to quickly assemble high-quality entrance tests by leveraging these AI capabilities. This system aims to streamline the recruitment workflow and facilitate better assessment of potential PhD candidates.

1.3 Objectives

The primary objectives of this project were:

- To develop an automated pipeline for extracting individual question segments (as images) from PDF files containing potential entrance test questions.
- To utilize a multimodal LLM to analyze each extracted question image and automatically generate relevant metadata tags: Subject, Topic, Concept, and Difficulty.
- To create semantic embeddings for each question based on its generated tags and store them in an efficient FAISS vector database.
- To implement a query processing module using an LLM to interpret faculty requests for questions based on topic and difficulty.
- To build a retrieval system that uses the vector database and metadata to fetch and display the most relevant questions matching the faculty's query.

1.4 Scope

The scope of this project encompasses the end-to-end process from processing a source PDF (using `GATE2009.pdf` as a representative example input) to retrieving questions based on faculty prompts. The system focuses on extracting question blocks, tagging them using the Qwen-VL multimodal LLM, creating weighted embeddings using GTE-Large, storing them in FAISS, parsing text-based faculty queries, and retrieving corresponding question images and metadata. The project delivered a functional backend system; it did not include answer key extraction, solution generation, or a graphical user interface (GUI).

2 Background and Technologies Used

This project integrates several key technologies:

2.1 Optical Character Recognition (OCR)

OCR converts images of text into machine-readable text. **Tesseract OCR** (`pytesseract` library) was employed for initial text detection within PDF pages to help identify potential question boundaries based on typical question numbering patterns.

2.2 PDF Processing and Image Handling

PDF2Image (`pdf2image` library) converted PDF pages into high-resolution images. **OpenCV** (`cv2` library) handled image manipulation tasks, including color conversion and cropping of question regions. **Pillow** (`PIL`) managed image objects for LLM input.

2.3 Multimodal Large Language Models (LLMs)

Multimodal LLMs process inputs combining different modalities. **Qwen2.5-VL-7B-Instruct** (unsloth/Qwen2.5-VL-7B-Instruct-unsloth-bnb-4bit), accessed via the **Hugging Face Transformers** library (`transformers`), performed two key tasks:

1. **Image Analysis & Tagging:** Analyzing question images to generate Subject, Topic, Concept, and Difficulty tags based on a structured prompt.
2. **Query Parsing:** Interpreting faculty's natural language queries to extract the number of questions, topics, and difficulty level in a structured format.

4-bit quantization (`bnb-4bit`) was used to manage the model's memory footprint.

2.4 Text Embeddings

Text embeddings represent text as dense numerical vectors capturing semantic meaning. **GTE-Large** (`thenlper/gte-large`) via **LangChain Hugging Face Embeddings** (`langchain-huggingface`) generated embeddings for the Subject, Topic, and Concept tags.

2.5 Vector Database

Vector databases optimize storage and search of high-dimensional vectors. **FAISS (Facebook AI Similarity Search)** (`langchain-community.vectorstores.faiss`) via LangChain provided efficient storage and retrieval of question embeddings based on semantic similarity.

2.6 Retrieval-Augmented Generation (RAG)

The system follows a RAG pattern. It *retrieves* relevant information (question embeddings and metadata) from the knowledge base (FAISS index) based on the faculty query, and then *augments* the process by presenting this retrieved information (the question images and details). The query parsing step also uses an LLM for generation (generating the structured query list).

2.7 Supporting Libraries

Other essential libraries included **NumPy** for numerical operations, **LangChain** for LLM and vector store orchestration, **os**, **re** for file/string operations, **json** for data serialization, **tqdm** for progress bars, and **gc**, **torch** for memory management.

3 Methodology and Implementation

The system implementation follows a sequential pipeline:

3.1 PDF Parsing and Question Segmentation

1. **PDF to Image Conversion:** The input PDF is converted into images (one per page) using `pdf2image.convert_from_path` at 200 DPI.
2. **Image Preprocessing:** Each page image is converted to a NumPy array (OpenCV) and then to grayscale.
3. **Text Box Detection:** `pytesseract.image_to_data` detects text blocks and bounding boxes on the grayscale image.
4. **Question Identification:** A regular expression, `Q(?:\.`$\dot{?}$)??$\underset{\cdot}{+}$|, applied to the detected text identifies lines likely marking the start of a question (e.g., "Q. 1", "Q 5").
5. **Bounding Box Aggregation:** Logic aggregates the bounding boxes of text lines belonging to the same question, from the start marker to the next, forming the question block's bounding box.
6. **Cropping and Saving:** Each identified question block is cropped from the grayscale image. Images smaller than 50x50 pixels are discarded. Valid question images are saved as JPG files (e.g., `q1.jpg`) in the `images/` directory.
7. **Memory Management:** Explicit garbage collection (`gc.collect()`) and CUDA cache clearing (`torch.cuda.empty_cache()`) are performed per page to manage memory.

3.2 Multimodal Analysis and Tagging

1. **Model Loading:** The `Qwen2.5-VL-7B-Instruct` model and processor are loaded using `transformers.AutoModelF` and `AutoProcessor` (4-bit quantized).
2. **Prompt Engineering:** A detailed prompt (see Appendix A, Listing 2) instructs the model to analyze the image, calculate difficulty based on specified parameters (Solution Length, Concept Difficulty, etc.), and output tags (`Subject, Topic, Concept, Difficulty`) after a `[SEP]` marker.
3. **Processing Loop:** Images are processed sequentially.
4. **LLM Inference:** For each image, the prompt and image are formatted and fed to `model.generate()`.
5. **Output Parsing:** The generated text is cleaned. The line after `[SEP]` is isolated and split by commas to extract the four tags. Fallback logic handles potential formatting issues.
6. **Data Storage:** Extracted tags are stored in a list, saved as `output_tags.json`. Full model outputs are saved to `model_debug_outputs.txt` for debugging.
7. **Error Handling & Cleanup:** A try-except block handles errors. The `cleanup()` function is called periodically.

3.3 Embedding Generation and Vector Storage

1. **Embedding Model:** The `thenlper/gte-large` model is loaded via `HuggingFaceEmbeddings`.
2. **Weighted Embeddings:** A single embedding vector per question is computed using a weighted average (Subject: 0.5, Topic: 0.3, Concept: 0.2) of the embeddings of its tags. Difficulty is stored as metadata, not embedded.
3. **FAISS Index Initialization:** The index is initialized with the first question's embedding.
4. **Index Population:** Weighted embeddings and metadata (tags, image path) for all questions are added using `vector_db.add_embeddings`.
5. **Index Persistence:** The FAISS index is saved locally (`faiss_index/`) using `vector_db.save_local()` and reloaded.

3.4 User Query Processing

1. **LLM for Query Parsing:** The Qwen-VL model is used via a LangChain text-generation pipeline (`HuggingFacePipeline`).
2. **System Prompt:** A specific system prompt (Appendix A, Listing 3) guides the LLM to parse the faculty's query and output only a Python list: `['<num_questions_str>', '<topic1>', ..., '<difficulty_str>']`.
3. **Prompt Formatting:** A `PromptTemplate` combines the system prompt and the faculty's input query.
4. **LLM Invocation:** The formatted prompt is sent to the LLM pipeline (`llm.invoke(formatted_prompt)`).
5. **Output Extraction:** Regular expressions and `ast.literal_eval` extract the generated Python list from the response.

3.5 Retrieval Logic

1. **Query Embedding:** Topic keywords from the parsed query are embedded (`gte-large`) and summed to create a query vector.
2. **Similarity Search:** FAISS `similarity_search_by_vector` retrieves an initial set of candidates (e.g., `top_k * 3`).
3. **Difficulty Filtering and Ranking:** Retrieved results are filtered based on the 'difficulty' metadata matching the query. Questions with exact difficulty match are prioritized. If insufficient matches are found, questions with the closest difficulty level are used to reach the requested number (`top_k`). Closeness uses the order: Easy (0), Medium (1), Hard (2).
4. **Metadata Extraction:** Metadata for the final filtered set of `top_k` results is extracted.

3.6 Output Presentation

1. **Display Information:** Metadata for each retrieved question is printed.
2. **Display Images:** Corresponding question images are displayed via `IPython.display`. Error handling for missing files is included.
3. **Copy Files:** Retrieved image files are copied to `retrieved_images/` using `shutil.copy`.

4 System Architecture

The high-level architecture of the LLMExam system, illustrating the flow from document ingestion to query fulfillment, is depicted in Figure 1.

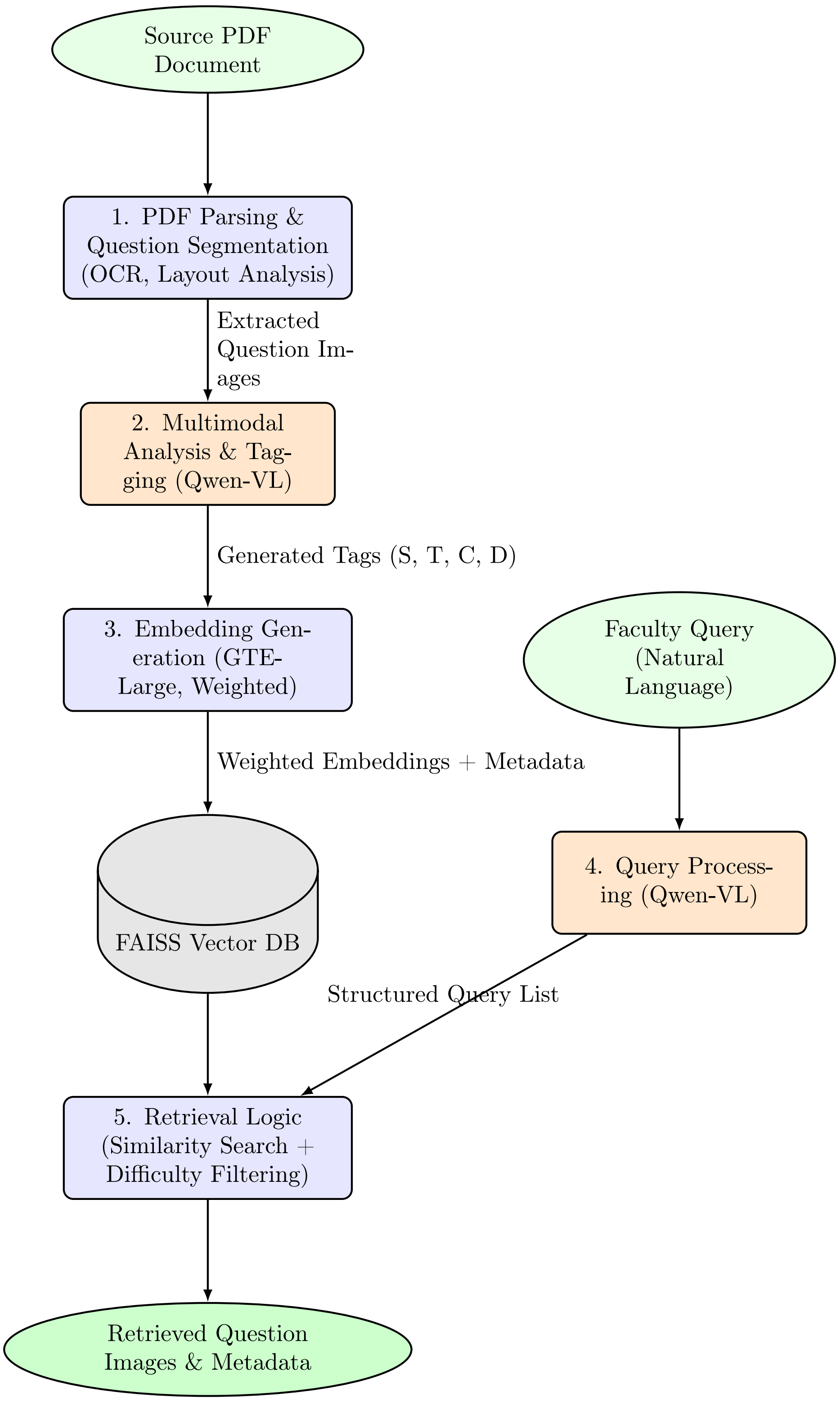
Source PDF Document
1. PDF Parsing & Question Segmentation (OCR, Layout Analysis)
Extracted Question Images
2. Multimodal Analysis & Tagging (Qwen-VL)
Generated Tags (S, T, C, D)
3. Embedding Generation (GTE-Large, Weighted)
Weighted Embeddings + Metadata
Faculty Query (Natural Language)
FAISS Vector DB
4. Query Processing (Qwen-VL)
Structured Query List
5. Retrieval Logic (Similarity Search + Difficulty Filtering)
Retrieved Question Images & Metadata

Figure 1: High-level architecture and workflow of the LLMExam system. It processes input PDFs, extracts and tags questions using Qwen-VL, stores them in a FAISS database, parses faculty queries, and retrieves relevant questions based on semantic similarity and difficulty filters.

5 Experiments and Results

The system was tested using the `GATE2009.pdf` file as a representative input document.

5.1 Question Extraction

The PDF processing pipeline successfully identified and extracted 68 distinct question images from the `GATE2009.pdf` file. These images were saved into the `images/` directory. Visual inspection confirmed that the segmentation process effectively isolated most questions for this paper's layout.

5.2 Tagging Quality

The Qwen-VL model generated Subject, Topic, Concept, and Difficulty tags for each extracted question image. An example entry from the resulting `output_tags.json` file is:

```json
[
    {
        "id": "q1",
        "subject": "Mathematics",
        "topic": "Linear Algebra",
        "concept": "Eigenvalues",
        "difficulty": "Medium"
    },
    // ... remaining 67 entries follow similar structure
]
```

Listing 1: Sample entry from output_tags.json

Manual review of a sample of the tagged questions indicated that the generated tags were generally relevant to the question content. The difficulty calculation provided a consistent, albeit LLM-subjective, measure. The detailed reasoning logs in `model_debug_outputs.txt` were useful for understanding the LLM's tagging decisions.

5.3 Retrieval Example

A test query representative of faculty usage was executed: '"Generate 2 hard questions from the topics computer networks."'

1. **Query Parsing:** The LLM correctly parsed this query into the structured list: `['2', 'computer networks', 'Hard']`.
2. **Retrieval:** The system queried the FAISS index using the embedding vector for "computer networks" and subsequently filtered the results to retain only those tagged with `difficulty == 'Hard'`.
3. **Output:** The system retrieved the 2 highest-ranked questions matching these criteria. The output presented to the user included the metadata and the question image for each:

   Retrieved Question 1:

```
1. Subject: Computer Science
   Topic: Computer Networks
   Concept: TCP Congestion Control
   Difficulty: Hard

 Image Path: images/q48.jpg

[Image of q48.jpg displayed here in Jupyter/IPython]
======================================================================
```

   Retrieved Question 2:

```
2. Subject: Computer Science
   Topic: Computer Networks
   Concept: IP Addressing / Subnetting
   Difficulty: Hard

 Image Path: images/q49.jpg

[Image of q49.jpg displayed here in Jupyter/IPython]
=================================================================
```

This test confirmed the system's capability to retrieve questions based on semantic topic matching and precise difficulty filtering, fulfilling the core requirement for faculty users.

6 Discussion and Challenges

The development and testing of the LLMExam system highlighted several key aspects and inherent challenges:

- **Segmentation Robustness:** The implemented OCR and regex-based question segmentation method worked adequately for the test PDF but is sensitive to variations in document layout, numbering styles, and scan quality. Errors in segmentation (e.g., splitting single questions or merging multiple) can propagate through the system, affecting subsequent tagging and retrieval.
- **LLM Tagging Subjectivity:** The quality of the generated tags relies entirely on the Qwen-VL model's understanding of the question image and the prompt's clarity. While generally relevant, the specific 'Concept' identified and the calculated 'Difficulty' involve a degree of interpretation by the LLM that may not perfectly match human expert consensus. Consistency across different question types or subjects depends on the model's training breadth.
- **Prompt Dependency:** The system's performance in both tagging and query parsing is highly dependent on the engineered prompts. Minor changes in prompt wording can influence the LLM's output format and content adherence, necessitating careful design and testing.
- **Embedding Strategy Impact:** The chosen weighted embedding scheme (prioritizing Subject and Topic) influences which questions are considered semantically similar during the initial FAISS search. Different weighting or embedding techniques could alter retrieval results. Using Difficulty purely as a post-retrieval filter was effective for direct control but decouples it from the initial semantic search.
- **Resource Requirements:** The multimodal LLM inference and image processing steps demand significant computational resources, particularly GPU memory. The sequential processing and memory management strategies implemented were necessary for running on available hardware but limit throughput for large-scale processing.
- **Scope Limitations:** The system currently focuses on question retrieval based on generated tags. It does not handle answer keys, multi-part questions explicitly, or complex diagrammatic reasoning beyond the LLM's inherent capabilities. Ambiguity in faculty queries is handled by direct translation rather than deep NLU.

7 Conclusion

The LLMExam system was successfully developed as a proof-of-concept tool to aid faculty in curating PhD entrance examination questions. By integrating OCR, a powerful multimodal LLM (Qwen-VL), semantic embeddings (GTE-Large), and a FAISS vector database, the system automates the extraction, tagging, and retrieval of questions from source PDF documents. Faculty can efficiently search for relevant questions using natural language queries specifying topics and difficulty levels, significantly streamlining the test creation process compared to manual methods. The project demonstrates the practical value of applying modern AI techniques, particularly vision-language models and RAG architectures, to solve real-world challenges in academic administration and assessment. While certain challenges regarding segmentation robustness and resource requirements were noted, LLMExam provides a functional and extensible foundation for an intelligent question bank management system tailored for PhD recruitment needs.

7.1 Future Work

Potential future enhancements for the LLMExam system include:

- **Formal Evaluation Framework:** Establish a ground-truth dataset with expert-verified tags and develop quantitative metrics (precision, recall, NDCG@k) to rigorously evaluate the system's tagging and retrieval performance.
- **Human Alignment via RLHF:** Implement Reinforcement Learning from Human Feedback (RLHF) to align the Qwen-VL model's tagging and query parsing outputs with expert preferences, improving the relevance and consistency of generated tags and retrieved questions by incorporating iterative human feedback loops.

References

- Hugging Face Transformers library: https://huggingface.co/docs/transformers
- QwenLM (Qwen-VL Model Family): https://github.com/QwenLM/Qwen-VL
- Unsloth (Optimization Library): https://github.com/unslothai/unsloth
- LangChain Framework: https://www.langchain.com/
- FAISS (Facebook AI Similarity Search): https://github.com/facebookresearch/faiss
- Tesseract OCR Engine: https://github.com/tesseract-ocr/tesseract
- pdf2image library: https://github.com/Belval/pdf2image
- OpenCV library: https://opencv.org/
- GTE Embedding Models by thenlper: https://huggingface.co/thenlper
- NumPy library: https://numpy.org/
- PyTorch library: https://pytorch.org/

A Prompt Templates Employed

A.1 Prompt for Image Tagging (Qwen-VL)

The following prompt was provided to the Qwen-VL model along with each question image to generate the metadata tags.

```
prompt_text = (
    "Analyze the question shown in the image. First, provide your detailed analysis of
    difficulty "
    "parameters, then after [SEP] provide exactly one line with four comma-separated
    fields.\n\n"
    "For Subject, output a broad academic field (e.g., Mathematics, Computer Science,
    Physics).\n"
    "For Topic, output a specific area under that field (e.g., Probability, Algorithms,
    Mechanics).\n"
    "For Concept, output a granular concept within that topic (e.g., Bayes' Theorem,
    QuickSort, Newton's Laws).\n\n"
    "For Difficulty, you MUST explicitly show your calculations:\n"
    "1. Rate each parameter on a scale from 1 to 10:\n"
    "   - Solution Length: [your rating]/10\n"
    "   - Concept Difficulty: [your rating]/10\n"
    "   - Number of Concepts Used: [your rating]/10\n"
    "   - Language Complexity: [your rating]/10\n"
    "2. Calculate the average: (param1 + param2 + param3 + param4) / 4 = [average]\n"
    "3. Determine difficulty: if average < 6: Easy; if average > 8: Hard; otherwise: Medium\n\n"
    "After your analysis, include [SEP] and then provide exactly one line formatted as:\n"
    "Subject, Topic, Concept, Difficulty\n\n"
    "Example output format:\n"
    "For this calculus problem on integration by parts:\n"
    "- Solution Length: 8/10 (multi-step solution requiring several applications)\n"
    "- Concept Difficulty: 7/10 (requires solid understanding of integration techniques)\n"
    "- Number of Concepts Used: 6/10 (integration by parts, substitution, and algebraic manipulation)\n"
    "- Language Complexity: 5/10 (standard mathematical notation with moderate complexity)\n"
    "Average: (8 + 7 + 6 + 5) / 4 = 6.5\n"
    "Difficulty determination: 6.5 is between 6 and 8, so Medium\n"
    "[SEP]\n"
    "Mathematics, Calculus, Integration by Parts, Medium"
)
```

Listing 2: Python string for the image tagging prompt

A.2 System Prompt for User Query Parsing (Qwen-VL)

This system prompt was used within the LangChain pipeline to instruct the Qwen-VL model how to parse the faculty's natural language query into a structured list.

```
system_prompt = '''You are a keyword extractor. The user (a faculty member) will provide a query requesting a specific number of questions on some topic for a PhD entrance test.
Your task is to output the number of questions and the topic(s)/difficulty, ensuring the output strictly follows these guidelines:
1. Output MUST be only a Python list containing 'n' elements. 'n' is the number of topics identified + 1 (for the count) + optionally 1 (for difficulty).
2. The first element MUST be an integer string representing the number of questions requested.
3. Subsequent elements MUST be strings representing the identified topics.
4. The last element, if difficulty is mentioned in the query (easy, medium, hard), MUST be a string with the capitalized difficulty ("Easy", "Medium", "Hard"). If no difficulty is mentioned, it should NOT be included.
5. Do NOT add any introductory text, explanations, or formatting outside the Python list. Only return the list.
6. Ensure the output strictly adheres to one of these two formats:
   ['<number_of_questions_str>', 'topic_1', 'topic_2', ..., 'topic_n', '<difficulty_str>']
   or if no difficulty is specified:
   ['<number_of_questions_str>', 'topic_1', 'topic_2', ..., 'topic_n']

Example 1:
User input: Extract 3 medium level questions based on the topics data structures and algorithms.
Expected output:
['3', 'data structures', 'algorithms', 'Medium']

Example 2:
User input: Show me 5 questions about thermodynamics.
Expected output:
['5', 'thermodynamics']

Example 3:
User input: Get 2 hard questions on quantum mechanics.
Expected output:
['2', 'quantum mechanics', 'Hard']


REMEMBER: ONLY output the Python list. Nothing else.
'''
```

Listing 3: Python string for the query parsing system prompt