**Descriptive Paper Evaluator - Prompt Structure**

**System Overview Prompt**

You are an advanced Descriptive Paper Evaluator system designed to assist instructors in evaluating student answer papers. You'll use LangChain for workflow orchestration, PyMuPDF/fitz for PDF processing, and a quadrant vector database for efficient semantic search. Your workflow includes:

1. Process instructor-uploaded materials (syllabus, textbooks, question papers)

2. Generate model answers for evaluation benchmarking

3. Analyze student handwritten answers (from scanned PDFs)

4. Evaluate answers using semantic textual similarity and plagiarism detection

5. Generate detailed feedback for both instructors and students

Focus on simple, accessible implementation using proven technologies like:

- LangChain for component integration

- PyMuPDF/fitz for PDF handling without OCR

- FAISS as lightweight vector databases

- Sentence-Transformers for embedding generation

- Streamlit for a simple user interface

**Input Processing Prompt**

For processing input documents, use LangChain document loaders with PyMuPDF/fitz:

1. SYLLABUS (PDF/JSON):

- Use PyMuPDF for PDF extraction or json.loads() for JSON

- Extract course objectives (COs), learning outcomes, and topic weightage

- Store in a simple dictionary structure or Pandas DataFrame

2. TEXTBOOKS (2-3 PDFs):

- Use PyMuPDF/fitz to extract text while preserving structure

- Use LangChain's TextSplitter to create chunks of appropriate size

- Embed chunks using a simple model like all-MiniLM-L6-v2

- Store in FAISS or Chroma vector database for efficient retrieval

3. QUESTION PAPER (XLSX/JSON):

- Use pandas for XLSX or json.loads() for JSON

- Parse question structure: question numbers, full text, marks allocation

- Extract evaluation rubrics for each question

- Store in a simple structured format for easy access

**Question Classification Prompt**

For each question in the question paper, use a lightweight classification approach:

1. Implement a simple rule-based classifier or fine-tuned DistilBERT to categorize questions:

- Definition/Concept explanation

- Problem-solving/Numerical

- Analysis/Critical thinking

- Compare and contrast

- Case study analysis

- Descriptive/Essay

- Short answer

2. For each question type, create a template prompt that will guide model answer generation

- Store these in a simple lookup dictionary

- Maintain consistent structure for prompts based on question type

**Model Answer Generation Prompt**

For each classified question, generate a model answer using a template-based approach:

1. Retrieve relevant content from textbook vector database using:

- Question text as query

- LangChain's similarity search with quadrant retrieval method

- Top-k most relevant chunks

2. Use a structured template based on question type:

Given the following question: "{question\_text}"

This is a {question\_type} question worth {marks} marks related to Course Objective {CO}.

The evaluation rubric includes: {rubric\_points}

Based on the following relevant content from the course materials: {retrieved\_content}

Generate a comprehensive model answer that:

* Fully addresses all components of the question
* Incorporates all elements from the evaluation rubric
* Aligns with Course Objective {CO}
* Is approximately {marks \* 150} words in length
* Uses proper terminology and structure for this subject area

3. Store the generated model answer in a simple database or JSON file for reference

**Student Answer Processing Prompt**

For processing student handwritten answers:

1. Use PyMuPDF/fitz to handle the scanned PDF documents:

- Extract page images without OCR

- Use basic image processing to identify page structure

- Look for handwritten question numbers using simple pattern recognition

2. For answer segmentation:

- Use PyMuPDF to identify text regions and possible question boundaries

- Apply rule-based segmentation based on:

\* Page numbers

\* Question number indicators

\* Visual spacing between sections

3. Create a simple data structure to map:

- Student ID → Answer document → Question number → Answer segment

- Store this mapping in a lightweight database like SQLite

**Answer Evaluation Prompt**

For each extracted student answer, use a straightforward evaluation pipeline:

1. Semantic Textual Similarity (STS) evaluation:

- Embed both model answer and student answer using Sentence-Transformers

- Use quadrant vector search to find matching concepts between answers

- Calculate similarity scores for each rubric point

- Use a simple scoring formula: (similarity\_score \* max\_marks)

2. Simple plagiarism detection:

- Compare embeddings against textbook content

- Use cosine similarity threshold to identify potential plagiarism

- Create a similarity matrix between student submissions

- Flag submissions with unusually high similarity

3. Calculate marks using a transparent formula:

- Base score from STS evaluation

- Adjustments based on rubric coverage

- Deductions for any identified plagiarism

- Store results in a simple CSV or SQLite database

**Instructor Feedback Generation Prompt**

Generate instructor feedback using a template-based approach:

1. Use pandas to create an overall class performance summary:

- Basic statistics (mean, median, range) per question

- Simple visualization of mark distribution

- CO achievement calculation using a straightforward formula

2. For individual student evaluation reports:

Student ID: {student\_id} Total Marks: {total\_marks}/{max\_marks}

Question-by-Question Breakdown: {for each question} Question {number} ({marks}/{max\_marks}):

* CO Alignment: {CO\_number}
* Rubric Performance:
  + {rubric\_point\_1}: {score\_1}/{max\_score\_1}
  + {rubric\_point\_2}: {score\_2}/{max\_score\_2} ...
* Content Coverage: {coverage\_percentage}%
* Plagiarism Check: {result}

Areas for Improvement:

* {identified\_gap\_1}
* {identified\_gap\_2} ... {end for}

3. Store all reports in a simple folder structure for easy access

**Student Feedback Generation Prompt**

Generate student feedback using a simple template:

1. Create a personalized report with straightforward metrics:

Performance Report for {student\_id}

Overall Score: {total\_marks}/{max\_marks} ({percentage}%) Class Percentile: {percentile}

Course Objective Performance:

* CO1: {co1\_score}/{co1\_max}
* CO2: {co2\_score}/{co2\_max} ...

Question-by-Question Feedback: {for each question} Question {number}:

* Your Score: {marks}/{max\_marks}
* Strengths:
  + {strength\_1}
  + {strength\_2}
* Areas for Improvement:
  + {weakness\_1}
  + {weakness\_2}
* Study Resources: See {textbook\_reference} {end for}

Key Focus Areas for Improvement:

1. {focus\_area\_1}
2. {focus\_area\_2}
3. {focus\_area\_3}

2. Export feedback as a simple PDF or HTML file

**Implementation Requirements Prompt**

Implement this system using these simplified technologies:

1. Core Processing:

- LangChain for orchestration

- PyMuPDF/fitz for PDF processing without OCR

- Sentence-Transformers (all-MiniLM-L6-v2) for embeddings

- FAISS or Chroma for vector storage with quadrant retrieval

2. Data Management:

- SQLite for structured data storage

- Simple file system for document management

- Basic authentication system for instructors and students

3. User Interface:

- Streamlit for a simple, accessible web interface

- Matplotlib or Plotly Express for basic visualizations

- Tabular display of results using Pandas DataFrames

4. Deployment:

- Local deployment option for privacy

- Simple Docker container for easy setup

- Minimal resource requirements (can run on CPU)

Help me build a system that works reliably with minimal technical complexity, while still achieving all functional requirements with modular approach .  
  
  
  
**1. document\_processor.py**

* **Purpose:** Handles syllabus, textbook, and question paper ingestion.
* **Core Features:**
  + PyMuPDF for PDF parsing.
  + FAISS for textbook vector storage.
  + Syllabus parsing from PDF/JSON.
  + Chunking with LangChain RecursiveCharacterTextSplitter.
  + Search via embedding similarity with optional book filtering.
  + CO analysis and question-topic matching.

**✅ 3. answer\_generator.py**

* **Purpose:** Generates and stores model answers using quadrant vector search.
* **Core Features:**
  + Quadrant-based semantic retrieval from textbook chunks.
  + Structured prompt building based on question metadata + rubric + COs.
  + Compatible with pluggable llm\_generate\_func() to support Mixtral, GEMINI\_API, etc.
  + Stores results in model\_answers.db.

**✅ 4. student\_processor.py**

* **Purpose:** Parses student handwritten answer PDFs.
* **Core Features:**
  + Uses PyMuPDF to extract text from scanned PDFs (no OCR).
  + Regex-based answer segmentation using visible question markers (e.g., Q1, 1., 1a).
  + Stores in student\_answers.db.
  + Registers new students and tracks question-to-answer mappings.

**✅ 5. evaluator.py**

* **Purpose:** Scores student answers against model answers.
* **Core Features:**
  + Semantic similarity scoring (SentenceTransformer).
  + Rubric point matching via segment-wise similarity.
  + Plagiarism detection from textbooks and peer answers.
  + Custom scoring logic: weighted similarity + rubric + plagiarism penalties.
  + Writes results to evaluations.db.

**✅ 6. feedback\_generator.py**

* **Purpose:** Generates detailed feedback for instructors and students.
* **Core Features:**
  + Aggregates student evaluation results.
  + Computes class-wide statistics.
  + Visualizes performance (matplotlib charts → base64 for UI/HTML).
  + Generates per-student feedback with CO-level analysis.

Everything is modular, local, resource-light, and matches your minimal-complexity design goals. You're in great shape to connect this into a LangChain pipeline + Streamlit UI for full integration.