System Design Document: Surgical Analysis RAG POC

Version: 1.0 **Date:** September 9, 2025

1. Introduction

1.1. Project Purpose

This document outlines the system design for a Proof of Concept (POC) of a multimodal, Retrieval-Augmented Generation (RAG) system. The primary goal of this system is to provide a data-driven analysis of surgical procedures by comparing events from a surgical video (including surgeon's narration) against a knowledge base of established medical best practices derived from surgical textbooks.

This project functions as an advanced data analytics pipeline, transforming raw, unstructured data (video, audio, PDF text) into structured, actionable insights. The potential benefits of such a system include enhancing surgical training by providing objective feedback, improving quality assurance through the identification of deviations from standard procedures, and creating novel, structured datasets for future medical research.

1.2. Scope

This document covers the complete architecture of the POC, including the offline data ingestion and model training pipelines, and the online web application for performing the analysis. It is intended to be a comprehensive technical guide for developers seeking to understand, replicate, and extend the current system.

In-Scope for this POC:

- End-to-end functionality from video upload to JSON analysis report.
- Use of specific open-source models (LLaVA) and APIs (Gemini).
- Deployment on a single, cloud-based GPU instance.
- A functional web interface for demonstration purposes.

Out-of-Scope for this POC:

- Real-time (live) video stream analysis.
- HIPAA compliance and patient data privacy protocols.
- A production-ready, multi-user, scalable deployment.
- Clinical validation or certification of the analytical output.

2. System Architecture Overview

The system is designed with a modular, two-phase architecture: an offline preparation phase and an online inference phase. This separation is a standard MLOps practice that allows the computationally expensive data preparation to be done once, making the online analysis component faster and more efficient.

Phase 1: Offline Knowledge Ingestion & Model Training

- **Objective:** To process raw surgical textbooks and train a custom AI model to create an intelligent, searchable knowledge base. This phase transforms unstructured text into a structured, semantically enriched asset.
- **Key Scripts:** `1_ingest_knowledge_base.py`, `3_train_ner_model.py`
- **Output:** A persistent vector database (`vector_db_enriched/`) and a fine-tuned classifier model (`surgical_ner_model/`).

Phase 2: Online Analysis & Web Application

- **Objective:** To provide a web interface for users to upload a video and receive a detailed, RAG-powered analysis. This phase handles the live data processing and insight generation.
- **Key Script:** `main_app.py`
- **Process:** The application takes a video, runs it through a multimodal perception pipeline (audio transcription and visual captioning), queries the knowledge base using the enriched data, and generates a final analytical report.

3. Phase 1: Knowledge Base Ingestion Pipeline

This pipeline prepares the surgical textbooks for intelligent retrieval. It is designed to be run manually whenever the knowledge base needs to be created or updated.

- **Script:** `1_ingest_knowledge_base.py`
- **Input:** A directory of PDF textbooks (`textbooks/`).
- **Output:** An enriched Chroma vector database (`vector_db_enriched/`).

3.1. Components & Technologies

1. **Document Loading:**
    - **Technology:** `langchain.document_loaders.PyMuPDFLoader`. This loader was chosen for its high performance and robustness in handling complex PDF layouts with tables and images.

    - **Process:** Iterates through the `textbooks/` directory and loads all PDF files. The text content and source metadata (filename, page number) are extracted from each page.
2. **Text Splitting:**
    - **Technology:** `langchain.text_splitter.RecursiveCharacterTextSplitter`. This splitter is ideal as it attempts to split text along semantic boundaries (newlines, sentences) before resorting to a hard character count.
    - **Process:** The raw text from all pages is split into smaller, overlapping chunks (1000 characters with a 150-character overlap) to create manageable documents for embedding and retrieval. The overlap ensures that concepts spanning two chunks are not lost.
3. **Metadata Enrichment (Classifier):**
    - **Technology:** Hugging Face `transformers.pipeline` loading the custom-trained `surgical_ner_model/`.
    - **Process:** Each text chunk is passed through the fine-tuned NER model. The model identifies and extracts key surgical entities according to the predefined schema (`INSTRUMENT`, `ANATOMY`, `ACTION`, `OBSERVATION`). These entities are added as new keys to the metadata of each document chunk (e.g., `metadata={'source': 'book.pdf', 'page': 42, 'ANATOMY': 'femoral artery'}`).
4. **Embedding:**
    - **Technology:** `langchain_community.embeddings.HuggingFaceEmbeddings` with the `all-MiniLM-L6-v2` model. This is a high-performance model that creates 384-dimensional vectors.
    - **Process:** The embedding model, configured to run on the GPU (`cuda`), converts the text content of each enriched chunk into a dense vector representation.
5. **Vector Storage:**
    - **Technology:** `langchain_community.vectorstores.Chroma`. Chroma is chosen for the POC due to its simplicity; it is a lightweight, local, file-based vector store that requires no separate server setup.
    - **Process:** The text chunks, their comprehensive metadata (source, page, and NER tags), and their corresponding vector embeddings are stored in a persistent directory (`vector_db_enriched/`).

4. Phase 2: NER Model Fine-Tuning

This pipeline creates the custom classifier used for metadata enrichment. It is a critical component for adding a semantic layer to the knowledge base.

- **Script:** `3_train_ner_model.py`

- **Input:** A hand-labeled JSON file (`ner_training_data.json`).
- **Output:** A fine-tuned NER model saved to the `surgical_ner_model/` directory.

4.1. Components & Technologies

1. **Input Data:**
    - **Format:** A JSON file containing a list of objects. Each object has a `"text"` field and a `"labels"` list. Each item in the `"labels"` list is a dictionary specifying the `start` and `end` character indices of the entity and its `label` (e.g., `INSTRUMENT`).
2. **Base Model:**
    - **Technology:** `distilbert-base-uncased` from Hugging Face. This model is chosen for its excellent balance of performance and computational efficiency, which allows for rapid fine-tuning.
3. **Training Orchestration:**
    - **Technology:** Hugging Face `transformers.Trainer` and `datasets` library.
    - **Process:**
        - The script loads the JSON data into a `Dataset` object.
        - It tokenizes the text using the base model's tokenizer and carefully aligns the character-based labels with the new sub-word tokens, creating IOB2-formatted tags (`B-INSTRUMENT`, `I-INSTRUMENT`, etc.).
        - The `Trainer` class, configured with hyperparameters (`learning_rate=2e-5`, `num_train_epochs=5`), handles the fine-tuning loop.
        - The final, best-performing model checkpoint is saved to disk, ready for use in the ingestion pipeline.

5. Phase 3: Online Inference & Web Application

This is the main, user-facing application that integrates all components into a single service.

- **Script:** `main_app.py`

5.1. Components & Technologies

1. **Web Server & API:**
    - **Technology:** `FastAPI` and `Uvicorn`. FastAPI is a modern, high-performance ASGI framework.
    - **Functionality:**
        - Serves the static `index.html` file at the root URL (`GET /`).

        - Provides a `POST /analyze_video` endpoint that accepts a multipart form data request containing the video file and two prompt strings.
        - Crucially, it uses `BackgroundTasks` to execute the long-running analysis pipeline, immediately returning a `202 Accepted` status to the user and preventing browser timeouts.
2. **Perception Pipeline (Executed in a Background Task):**
    - **Audio Extraction:**
        - **Technology:** `ffmpeg` (system-level dependency).
        - **Process:** Called via Python's `subprocess` module. The command `ffmpeg -i '{video_path}' -vn -acodec copy -y '{audio_path}'` is used to efficiently extract the audio stream without re-encoding, ensuring speed and quality.
    - **Audio Transcription:**
        - **Technology:** `google-generativeai` library calling the `gemini-1.5-flash` model.
        - **Process:** The temporary audio file is uploaded to the Google AI API. The model is prompted to provide a verbatim transcript, which is returned as a single block of text.
    - **Visual Analysis:**
        - **Technology:** A local `llava-hf/llava-1.5-7b-hf` model running via the Hugging Face `transformers` library.
        - **Process:** The script samples one frame every 5 seconds from the video using OpenCV. Each frame is sent to the local LLaVA model on the GPU, which is loaded in 4-bit precision (`load_in_4bit=True`) to minimize memory usage. The model generates a text description of the observed surgical action.
3. **RAG (Retrieval-Augmented Generation) Core:**
    - **Query Formulation:** The full audio transcript and all generated visual descriptions are combined into a single, comprehensive text query to provide maximum context to the retrieval system.
    - **Hybrid Retrieval:** This is a two-stage process for high-precision retrieval.
        - **Metadata Filtering:** The custom NER model (`get_ner_pipeline()`) first processes the query to extract entities (e.g., `ANATOMY: "pedical"`).
        - **Semantic Search:** A `Chroma` retriever then performs a search within the vector database. It is configured to *first* filter the search space to only include documents that contain the identified metadata tags, and *then* it finds the top 5 most semantically similar chunks from that filtered subset.
    - **Generation:**
        - **Technology:** `google-generativeai` library calling the `gemini-1.5-flash` model.

        - **Process:** A final, detailed prompt is constructed. It includes the user's editable System Prompt, the retrieved textbook context, the full audio transcript, the visual descriptions, and the user's editable User Prompt. The Gemini model generates the final analysis based on this complete context.
4. **Output:**
    - The API immediately returns a `202 Accepted` response to the user's browser, confirming the task has started.
    - The background task, upon completion, saves a final, structured `analysis_output.json` file to the `outputs/` directory on the server, ready for download via `scp`.

5.2. Deployment & Execution Environment

- **Host:** RunPod GPU Cloud Instance (e.g., NVIDIA RTX A5000) running a base Ubuntu 22.04 image.
- **Environment Manager:** `Miniconda`. The Conda installation itself and the project's Python environment (`surgical-poc`) are stored in the persistent `/workspace` directory to survive pod restarts.
- **System Dependencies:** `ffmpeg`. This is a system-level tool and must be reinstalled (`apt-get install ffmpeg`) on each pod restart due to the ephemeral nature of the root filesystem.
- **Startup Procedure:** Requires a specific sequence of commands to initialize Conda, install system dependencies, activate the Python environment, set API keys, and launch the Uvicorn web server.