# CVA EXPERIMENTATION TECHNICALITIES

Version: 1.0

Date: July 16, 2024

Prepared by : Vidhya Varshany J S

Status: In Progress

# Table of Contents

[CVA Experimentation Technicalities 2](#_Toc203681503)

[1. Executive Summary & System Purpose 4](#_Toc203681504)

[1.1. Overview 4](#_Toc203681505)

[1.2. Business Problem 4](#_Toc203681506)

[1.3. Solution 4](#_Toc203681507)

[2. System Architecture 5](#_Toc203681508)

[2.1. High-Level Architecture Diagram 5](#_Toc203681509)

[2.2. Technology Stack 5](#_Toc203681510)

[3. Core Components & Technical Deep Dive 6](#_Toc203681511)

[3.1. src/indexer.py: The Data Indexer 6](#_Toc203681512)

[3.2. src/retriever.py: The Core Logic Engine 7](#_Toc203681513)

[3.2.1. MedicalRecordRetriever Class 7](#_Toc203681514)

[3.2.2. Context Retrieval & Expansion 7](#_Toc203681515)

[3.2.3. Prompt Engineering 7](#_Toc203681516)

[3.2.4. Question Processing & Output Generation 8](#_Toc203681517)

[3.3. Medical\_Record\_Analysis.ipynb: The Orchestration Notebook 8](#_Toc203681518)

[4. Data Flow & Workflow 9](#_Toc203681519)

[4.1. Phase 1: Indexing Workflow (One-Time) 9](#_Toc203681520)

[4.2. Phase 2: Retrieval & Analysis Workflow (Per Run) 9](#_Toc203681521)

[5. Setup & Usage Guide 10](#_Toc203681522)

[5.1. Prerequisites 10](#_Toc203681523)

[5.2. Installation 10](#_Toc203681524)

[5.3. Running the System 10](#_Toc203681525)

[Future Enhancements 11](#_Toc203681526)

# 1. Executive Summary & System Purpose

## 1.1. Overview

This document provides a comprehensive technical overview of the Clinical Validation Audit as an Automation System. This system is a sophisticated Retrieval-Augmented Generation (RAG) pipeline designed to automate the process of answering complex clinical questions based on large volumes of unstructured medical records. It leverages a combination of advanced search technology and Large Language Models (LLMs) to provide accurate, synthesized, and evidence-based answers for clinical audits.

## 1.2. Business Problem

Clinical auditing is a time-consuming and manual process that requires skilled professionals to read through thousands of pages of medical records to find specific data points and evidence. This process is prone to human error, inconsistency, and significant delays. The primary business challenge is to accelerate this process, improve the accuracy of audits, and free up valuable clinical resources for patient care.

## 1.3. Solution

The system addresses this challenge by implementing a two-phase automated pipeline:

**Indexing:** All source medical record data (from CSV files) is pre-processed and ingested into a high-performance Elasticsearch index. This makes the entire corpus of medical data instantly searchable.

**Retrieval & Analysis:** For a given set of audit questions, the system intelligently retrieves the most relevant text snippets from the indexed data. This context is then provided to a Large Language Model (LLM), which is guided by highly specific "prompts" to synthesize answers, create tables, and generate professional reports in both Excel and PDF formats.

# 2. System Architecture

## 2.1. High-Level Architecture Diagram

A diagram of a diagram

AI-generated content may be incorrect.

## 2.2. Technology Stack

Programming Language: Python 3.9+

**Core Libraries:**

**pandas:** For data manipulation and handling of CSV/Excel files.

**elasticsearch-py:** The official Python client for interacting with Elasticsearch.

**openai:** The client for communicating with the Large Language Model API.

**Reportlab:** A pure Python library for generating high-quality PDFs from Markdown text.

**Search & Retrieval:**

**Elasticsearch:** A powerful, scalable search engine used for fast, fuzzy-text retrieval of medical context.

**Large Language Model:**

Llama 3.1 (8B Instruct): The LLM responsible for understanding context and generating synthesized answers.

**Orchestration & Development Environment:**

Jupyter Notebook: Used as the primary interface for running analyses, experimenting, and generating reports.

# 3. Core Components & Technical Deep Dive

## 3.1. src/indexer.py: The Data Indexer

This script is responsible for the one-time setup of the search index.

**ElasticsearchIndexer Class:** Encapsulates all logic for connecting to ES, defining the index structure, and ingesting data.

**create\_index():** Defines the "schema" or mapping for the index. This tells Elasticsearch how to analyze the data. For example, the content field is treated as searchable text, while audit\_id is a keyword for exact filtering.

**prepare\_documents():** Reads the arf\_demo\_merged.csv file row by row, converting each row into a JSON document suitable for Elasticsearch. The original DataFrame index is preserved as the document \_id for easy lookups.

**index\_documents():** Uses the efficient bulk helper to send documents to Elasticsearch in large batches, which is significantly faster than one-by-one ingestion.

## 3.2. src/retriever.py: The Core Logic Engine

This is the most critical file, containing the main MedicalRecordRetriever class.

### 3.2.1. MedicalRecordRetriever Class

This class is the central "brain" of the application. Upon initialization, it loads all necessary data (CSV and Excel) and establishes connections to both Elasticsearch and the LLM API.

### 3.2.2. Context Retrieval & Expansion

**retrieve\_context\_for\_atoms():** This is the primary entry point for retrieval. It acts as a router.

If use\_elasticsearch is True, it calls \_elasticsearch\_search().

If False, it calls the fallback \_csv\_based\_search().

**\_elasticsearch\_search():** Constructs a sophisticated JSON query for Elasticsearch. It uses a bool query to combine multiple match clauses with an OR logic (minimum\_should\_match: 1) for the atoms, while filtering by target\_classes and audit\_ids with an AND logic.

**\_csv\_based\_search():** A robust, pure-Python fallback. It uses fast, vectorized pandas operations (.str.contains()) to first filter the DataFrame by class/audit and then search for the atoms within the content column using regex.

**\_expand\_context\_around\_row():** This crucial function takes a single hit (identified by its original row index). It finds all text lines on the same page, sorts them by their vertical (y) coordinate to simulate reading order, and extracts the text from 2 lines above and 2 lines below the hit. This creates a rich, coherent paragraph of context.

### 3.2.3. Prompt Engineering

**build\_sanitized\_context\_for\_llm():** This method prepares the context that will be sent to the LLM. It deliberately strips all metadata (page numbers, class names) to prevent the LLM from leaking this information in its answer. It also manages the token limit by dropping the least relevant context chunks if the prompt gets too long.

**\_generate\_structured\_prompt():** This is the prompt engineering core. It constructs a highly specific, multi-part prompt:

* **System Prompt:** A global set of rules defining the LLM's persona as a factual, non-conversational medical analyst.
* **Context Block:** The sanitized clinical context is wrapped in tags to clearly separate it from instructions.
* **Task Instructions:** A direct command to the LLM based on the generation\_type. The instructions are forceful and explicit (e.g., "Your entire response must be ONLY...", "If data is not present, your entire response must be the single sentence...").

### 3.2.4. Question Processing & Output Generation

**process\_question():** Orchestrates the entire workflow for a single question.

It retrieves the context using **retrieve\_context\_for\_atoms()**.

It splits multi-part questions (e.g., "1. ..., 2. ...") into a list of sub-questions.

It loops through each sub-question, generates a prompt, and calls the LLM using **\_call\_llm().**

It assembles the responses from sub-questions into a single, formatted string.

It returns a comprehensive dictionary containing all the metadata for the final Excel report.

**create\_pdf\_report():** This method uses the reportslab library. It constructs a single large Markdown string containing all the headers, questions, and LLM responses. It then intelligently splits the responses for multi-part questions to display them separately. Finally, it calls the library to convert this Markdown string into a high-quality, professionally formatted PDF, with excellent support for tables.

## 3.3. Medical\_Record\_Analysis.ipynb: The Orchestration Notebook

This Jupyter Notebook serves as the high-level user interface for the system. Its role is to:

* Configure: Set all paths, API keys, and analysis parameters (like TARGET\_AUDIT\_IDS and OUTPUT\_FORMAT).
* Orchestrate: Import the MedicalRecordRetriever class and call its methods to run the analysis.
* Analyze & Visualize: Contains a dedicated section for interactively comparing retrieval methods for a single question, allowing for rapid debugging and validation.
* Present: Uses IPython.display.Markdown to present outputs in a clean, readable format within the notebook.

## 4. Data Flow & Workflow

The system operates in two distinct phases.

## 4.1. Phase 1: Indexing Workflow (One-Time)

**Trigger:** The run\_indexing() function is executed in the notebook.

**Instantiate:** An ElasticsearchIndexer object is created.

**Delete & Create:** The old Elasticsearch index (if it exists) is deleted, and a new one is created with the updated mapping.

**Prepare:** The prepare\_documents() method reads the entire source CSV and converts each row into a JSON document.

**Ingest:** The index\_documents() method sends these documents in bulk to Elasticsearch.

**Result:** The Elasticsearch index is fully populated and ready for searching.

## 4.2. Phase 2: Retrieval & Analysis Workflow (Per Run)

**Trigger:** The run\_analysis\_pipeline() function is executed in the notebook.

**Instantiate:** A MedicalRecordRetriever object is created, loading all data and connecting to services.

**Outer Loop (Audit IDs):** The system iterates through the list of TARGET\_AUDIT\_IDS.

**Inner Loop (Questions):** For each audit ID, it iterates through every question in the loaded Excel file.

**Retrieval:** process\_question() calls retrieve\_context\_for\_atoms(). This function applies all filters (audit ID, class) and finds matching text snippets using either Elasticsearch or CSV search.

**Expansion:** For each hit, \_expand\_context\_around\_row() is called to get surrounding lines of text.

**Context Building:** Two context strings are created: a sanitized one for the LLM and a rich one for logging. The token limit is enforced.

**Question Splitting:** The main question text is split into sub-parts if necessary.

**LLM Call:** For each sub-question, a prompt is generated and sent to the LLM.

**Response Assembly:** The answers to sub-questions are combined into a single formatted response string.

**Data Collection:** The final results for the question are stored in a list of dictionaries.

**Output Generation:** After all questions for an audit ID are processed, the system calls create\_pdf\_report() and/or saves an Excel file based on the OUTPUT\_FORMAT setting.

Loop: The process repeats for the next audit ID.

# 5. Setup & Usage Guide

## 5.1. Prerequisites

* Python 3.9+
* Access to an Elasticsearch instance.
* Access to an OpenAI-compatible LLM API endpoint and a valid API key.

## 5.2. Installation

Clone the project repository.

Set up a Python virtual environment.

Install the required Python packages:

**pip install pandas openpyxl elasticsearch-py openai**

## 5.3. Running the System

**Configure:** Open Medical\_Record\_Analysis.ipynb and update the configuration variables in the first cell (file paths, API keys, ES host).

**Run Indexer (If Needed):** If this is the first time running the system or if the source CSV has changed, uncomment and run the cell for "Run the Data Indexer".

**Initialize Retriever:** Run the cell to initialize the MedicalRecordRetriever. Check for success messages.

**Run Full Pipeline:** Configure the TARGET\_AUDIT\_IDS and OUTPUT\_FORMAT variables in the final cell, then execute it to generate the reports. The output files will be saved in the /output directory.

**Interactive Analysis:** Use the "Compare Retrieval Methods" cell to debug or analyze specific questions by changing the QUESTION\_INDEX\_TO\_TEST variable and re-running the cell.

# Future Enhancements

While the current system is robust, several state-of-the-art techniques can be integrated to further improve performance and accuracy. These ideas are parked for future development cycles.

Re-Ranking with Cross-Encoders…..