**REPORT ON DATA ENGINEERING ASSIGNMENT**

### Abstract:

The report details the implementation of a Python-based data pipeline designed to ingest, process, and analyze a subset of the LIDC-IDRI dataset, as per the given assignment. The pipeline automates the downloading of DICOM files, extracts the data, and validates each DICOM file which includes Key metadata like Patient ID, Study/Series UIDs, slice thickness, and pixel spacing, is parsed from each DICOM header using the **pydicom** library. The extracted metadata is then organized, loaded into a Pandas DataFrame for analysis, and persisted in a local SQLite database with a defined schema. Finally, the pipeline generates a summary report that includes key statistics and visualizations, such as the distribution of slice thickness, providing an initial overview of the dataset's characteristics.

**1. Project Overview and Objectives**

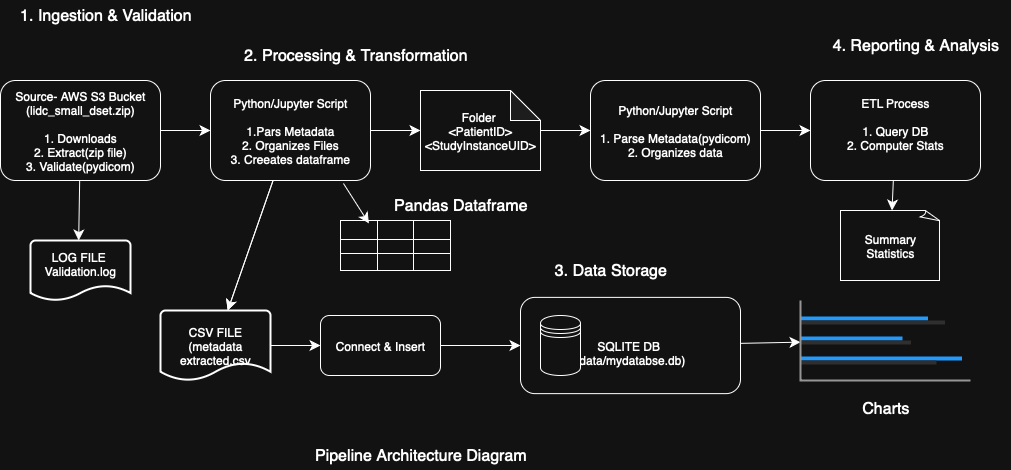
#### 1.1 Introduction

The goal of this project was to build an end-to-end pipeline to process a dataset of 10 de-identified LIDC CT scans provided in DICOM format. The objective was to demonstrate proficiency in handling medical imaging data, with a focus on ingestion, metadata extraction, structured storage, and basic reporting. The entire workflow was implemented in a Jupyter Notebook using Python and standard data science libraries.

#### 1.2 Key Objectives

The primary objectives of this assignment were as follows:

* **Data Ingestion:** Programmatically download all CT scan data from the provided S3 bucket and validate the integrity of the downloaded files.
* **Metadata Extraction:** Parse the DICOM headers of each slice to extract relevant fields such as Patient ID, Study/Series Instance UID, slice thickness, and pixel spacing.
* **Data Organization:** Reorganize the raw files into a logical folder structure based on patient and study identifiers.
* **Data Storage:** Design a minimal SQL schema and populate a local SQLite database with the extracted metadata.
* **Reporting:** Generate summary statistics, including the total number of studies and slices, and create visualizations to illustrate metadata distributions.
  1. **Architecture Diagram**



### 2. Methodology and Implementation

The pipeline was implemented as a series of sequential steps in a [Jupyter Notebook](https://colab.research.google.com/drive/1Et9kkZkvWWc0wee8sC4VvoJhDX77zFLj?usp=sharing), covering ingestion, validation, processing, storage, and reporting.

#### 2.1 Step 1: Data Ingestion and Validation

The first phase involved retrieving the data and ensuring its quality.

1. **Download:** The dataset, a single zip archive, was downloaded programmatically from the provided S3 pre-signed URL. A helper function was created using the **requests** library to stream the file, handle potential HTTP errors, and display a progress bar with **tqdm.**  
     
   Python  
   # Snippet: Function to download the dataset with progress bar and validation  
   def download\_file(url, out\_path):  
    try:  
    r = requests.get(url, stream=True, timeout=60)  
    r.raise\_for\_status()  
    total = int(r.headers.get("content-length", 0))  
    with open(out\_path, "wb") as f, tqdm(...) as pbar:  
    for chunk in r.iter\_content(1024 \* 16):  
    f.write(chunk)  
    pbar.update(len(chunk))  
    logging.info(f"Downloaded successfully: {out\_path}")  
    return True  
    except Exception as e:  
    logging.exception(f"Download failed: {e}")  
    return False
2. Extraction: The downloaded **lidc\_small\_dset.zip** archive was extracted using Python's built-in **zipfile** library into an **lidc\_extracted** directory. Error handling was included to catch corrupted zip files.

1. **Validation:** After extraction, each file was validated to confirm it was a well-formed DICOM file. This was achieved by attempting to read each file's header with  
   **pydicom.dcmread(..., stop\_before\_pixels=True),** which is a fast way to check metadata integrity without loading the heavy pixel data.  
   * **Result:** The validation process confirmed **2696 valid DICOM files** and **0 invalid files**, indicating a clean initial dataset. A [**dicom\_validation\_report.csv**](https://github.com/Tripti83/qure.ai_data_engineering_assignment/blob/main/logs/dicom_validation_report.csv) was generated for auditing purposes.

#### 2.2 Step 2: Metadata Extraction and Organization

With the data validated, the next step was to parse and structure the metadata.

1. **Parsing:** The pipeline iterated through all 2696 valid DICOM files and parsed their headers using pydicom. Key metadata fields were extracted using getattr for safe access, with default values assigned for any missing tags.  
     
   Python  
   # Snippet: Core metadata extraction loop  
   for f in tqdm(all\_dicoms, desc="Extracting metadata"):  
    try:  
    ds = pydicom.dcmread(str(f), stop\_before\_pixels=True, force=True)  
    except Exception:  
    continue # Skip invalid files  
     
    # Extract fields (safe defaults)  
    patient\_id = getattr(ds, "PatientID", "UNKNOWN\_PATIENT")  
    study\_uid = getattr(ds, "StudyInstanceUID", "UNKNOWN\_STUDY")  
    series\_uid = getattr(ds, "SeriesInstanceUID", "UNKNOWN\_SERIES")  
    slice\_thickness = getattr(ds, "SliceThickness", None)  
    # ... other fields ...  
     
    metadata\_records.append({ ... })

1. **Organization:** Concurrently, the files were copied into a new, logical directory structure: data/organized/<PatientID>/<StudyUID>/<SeriesUID>/. This makes the data easier to navigate and query at the file-system level.
2. **Transformation:** The list of extracted metadata dictionaries was loaded into a Pandas DataFrame. The number of slices per series was calculated using a groupby() operation and merged back into the main DataFrame, adding a useful aggregate metric (num\_slices) to each record.The final DataFrame was saved [metadata\_extracted.csv](https://github.com/Tripti83/qure.ai_data_engineering_assignment/blob/main/data/metadata_extracted.csv) for persistence.

#### 2.3 Step 3: Data Storage

A local SQLite database was chosen for structured storage, as it is lightweight and sufficient for this task.

1. **Schema Design:** A single table, dicom\_metadata, was designed to store the flat metadata structure.The schema was defined with appropriate data types (e.g., TEXT for UIDs, REAL for numeric values).  
     
   SQL  
   -- Snippet: SQL Schema Definition (DDL)  
   CREATE TABLE IF NOT EXISTS dicom\_metadata (  
    patient\_id TEXT,  
    study\_instance\_uid TEXT,  
    series\_instance\_uid TEXT,  
    sop\_instance\_uid TEXT,  
    slice\_thickness REAL,  
    pixel\_spacing REAL,  
    study\_date TEXT,  
    acquisition\_date TEXT,  
    instance\_number INTEGER,  
    file\_path TEXT,  
    num\_slices INTEGER  
   );
2. **Population:** The database table was populated by iterating through the Pandas DataFrame and executing an INSERT statement for each row. Upon completion, the database contained 2696 records, one for each DICOM slice.

[**metadata**.db](https://github.com/Tripti83/qure.ai_data_engineering_assignment/blob/main/data/mydatabase.db)

### 3. Summary Report and Visualizations

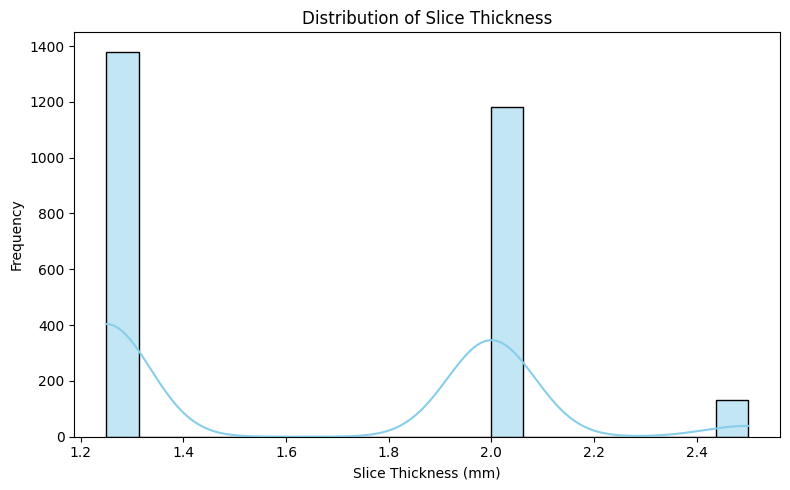
The final step was to query the processed data to generate summary statistics and visualizations as required.

#### 3.1 Summary Statistics

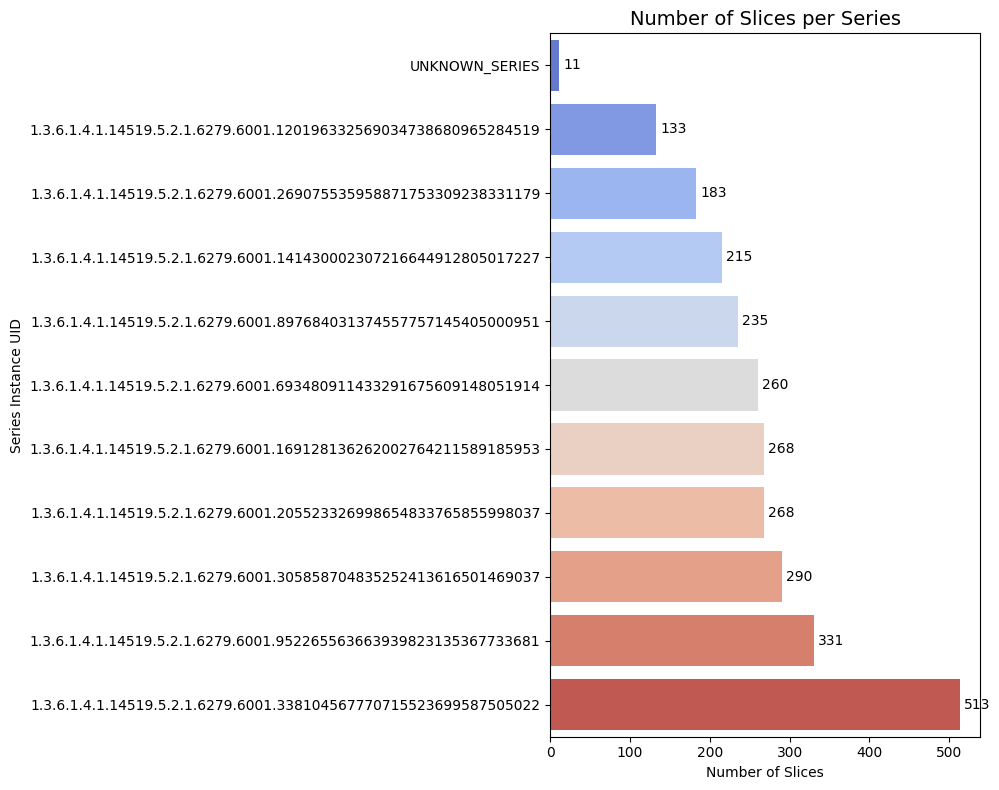
* **Total number of studies:** 10
* **Total slices across all scans: 2696**
* **Average number of slices per study:** 240-250
* **Slice Thickness Distribution:**
  + **Mean:** 1.64 mm
  + **Min:** 1.25 mm
  + **Max:** 2.50 mm

#### 3.2 Visualizations:

Visualizations were created using matplotlib and seaborn to provide a clearer understanding of the data's properties.



***Figure 1: Distribution of Slice Thickness (mm).*** The histogram shows that the most common slice thicknesses are 1.25 mm and 2.5 mm, which are typical values for CT imaging protocols.



***Figure 2: Number of Slices per DICOM Series.*** This chart illustrates the significant variation in the number of slices contained within each series, ranging from just a few dozen to over 500.

### Scalability and Monitoring Considerations

As an optional task, the following outlines how this pipeline could be adapted for a production environment with a much larger dataset (1,000+ scans).

* **Scalability:** To handle thousands of scans, the current single-threaded script would be re-architected for parallel processing. An ideal cloud-based approach would involve:  
  1. **Event-Driven Ingestion:** Use AWS S3 events to trigger an AWS Lambda function or a Batch job whenever a new scan archive is uploaded.
  2. **Parallel Processing:** Each scan would be processed in its own containerized environment, allowing for concurrent metadata extraction across hundreds of scans.
  3. **Scalable Storage:** Replace SQLite with a managed database service like Amazon RDS (for PostgreSQL) or a NoSQL database like DynamoDB to handle high-volume concurrent writes and reads.

* **Monitoring & Logging:** In a production setting, robust monitoring is crucial.  
  1. **Centralized Logging:** All logs (application logs, errors, metrics) would be sent to a centralized service like AWS CloudWatch Logs.
  2. **Metrics and Alarms:** Key performance indicators (KPIs) like processing throughput (scans/hour), error rates, and average processing time would be tracked as CloudWatch Metrics. Alarms would be configured to notify the team of critical failures (e.g., >5% error rate)
  3. **Error Handling:** Implement automated retries with exponential backoff for transient issues (e.g., network failures) and a dead-letter queue (DLQ) to store scans that fail processing repeatedly for manual inspection.

**References**

1.pydicom Documentation – Dataset basics: read, access, modify, write (using dcmread to load DICOM files).

2. Python pandas and sqlite3 libraries (Official Documentation).

3. Progress and logging in Python (logging module documentation).

4. Lots of googling!