****

**BACHELOR OF SCIENCE IN SOFTWARE DEVELOPMENT**

**FINAL YEAR PROJECT 1 : BSD 3106**

**REALTIME MEDICAL IMAGE ANALYZER**

**MAKINDU TRAUMA CENTER**

**BY**

**FESTUS MAITHYA KILONZO**

**SYSTEM DESIGN DOCUMENT SUBMITTED IN PARTIAL**

**FULFILLMENT FOR THE REQUIREMENTS FOR THE AWARD OF A DEGREE**

**IN SOFTWARE DEVELOPMENT**

**PRESENTED TO : DR. LUCY MBURU**

**Declaration**

I declare that this project is my original work and has not been presented in any other college or University for the award of a Diploma or Degree.

Student Details:

Name: …FESTUS MAITHYA KILONZO…… Date : …………22/10/2024………

Signature ………FMK………

Supervisor Details:

Name: ………DR LUCY MBURU……...…… Date : ………………………………

Signature …………………..………

**TABLE OF CONTENTS**

1.0 INTRODUCTION…………………………………...……………….…..…………………..5

1.2 INTENDED AUDIENCE…………… ……………………...…….………………………...5

1.3 AIMS OF SRS ………………………………………………….……………………...…......5

1.4 PROJECT SCOPE………………………………………………………………...………..…5

1.5 REFERENCE …………………………………………….…………………...………………5

1.6 OVERAL DESCRIPTION ……………………………………...……...………….…………5

1.6.1 Product Perspective…………………………………………...…………………………….5

1.7 PRODUCT FEATURES…………………………………..……………… ………………….7

1.7.1 Data Capturing………………………………………………………………………………7

1.7.2 Data Processing…………………………………………………………………...…………7

1.7.3 Data Output………………………………………………………………………….………7

1.7.4 Data Validation and Integrity…………………………………………..……………………7

1.7.5 Monitoring………………………………………………………………..…………………7

1.8 USER CLASSES & CHARACTERISTICS ……………………………….…………………7

1.9 OPERATING ENVIRONMENT ………………………………………………………..……8

1.9.1 Hardware……………………………………………………………………………….……8

1.9.2 Software………………………………………………………………………..……………8

1.10 DESIGN AND IMPLEMENTATION CONSTRAINTS……………………………………8

**1. INTRODUCTION**

This System Design Document outlines the design for a Real-Time Medical Image Analyzer which is a system in which users can upload medical images to be analyzed in real time, powered by the strength of pre-trained models coming from Google AI (Gemini). The system stitches together an inviting Streamlit frontend, a robust Django backend, together with an overtly scalable real-time data pipeline realized with Kafka and PySpark for message and task scheduling. The system architecture is made in such a way that it supports cloud deployment on GCP using Docker and Pulumi, scalable, maintainable, and secure.

**1.1 Purpose**

This system is aimed to democratize access to advanced medical image analysis by providing a user-friendly platform with rapid and accurate results without the need for specialized hardware or software. Emphasis on real-time performance reduces one of the most vexing delays in analyses that are often critical in time-sensitive medical contexts. This tool allows healthcare professionals to diagnose patients, hence providing better care to them.

**1.2 Scope**

The scope for the project includes extensive system architecture, data flow, user interface, API interactions, database schema, security considerations, designs of the data pipeline using Kafka and PySpark, a strategy for deployment using Docker and Pulumi, and a roadmap for scaling and enhancements in the future. This showcases a blueprint of development and deployment, marrying the key technical considerations with a roadmap to future growth of my project.

**1.3 System Design Constraints**

As the designer of this Real-time Medical Image Analyzer, I am aware of several constraints that may impact three key aspects involved in product development i.e. development, deployment, and performance. These might be due to real-time aspects, user demands, security, and dependency on APIs. Success in the project will come to a large extent from the capability of dealing effectively with these constraints.

Some of the major System Design Constraints are:

* **Real-time Analysis** -The Gemini API calls and LLM responses need to complete so fast that there is almost no time difference to the user. This is because some of the sessions from Streamlit to API will run concurrently with optimized requests for analysis and their feedback ,hence, latency becomes critical.
* Accuracy Requirements - Medical functions must have a high degree of diagnostic accuracy, therefore, the choice and evaluation of the model will be very critical.
* **Data Privacy and Security** - Strict security for sensitive data like user credentials and medical images need tight security thus giving need for secure authentication and communication protocols.
* **API Rate Limits** - Gemini and LLM APIs most probably have usage limits, rate limits, and costs hence efficient task scheduling and queues should be set up through pipelines and backend services. Since many models can also reduce the speed at which output responses are produced, thereby increasing resource consumption due to session timeouts, many requests will be on streaming, which is another point to consider if a caching mechanism with a timeout will be implemented. The implications for this will be that of efficient queuing, model request dispatch, and response streams aggregation, with minimum data persistence or storing on resource-limited environments at runtime for better-optimized real-time outputs.
* **Scalability** - The system shall support unlimited numbers of users and added image uploads with no form of performance degradation by using streaming tools like Apache Kafka and Spark, whereby I shall intensively implement asynchronous processes with queues.
* **User-Friendly Interface** - The front-end of the Streamlit application should be user-friendly, not overwhelming for the end user, and should provide clarity in flow and responsiveness. This relies on using libraries implemented within an app for browser caching and other mechanisms using the underlying architecture of Streamlit through its features
* **Cost Optimization** – The system performance needs to be weighed against the cost of the cloud resources involved, the usage of the APIs, computer instances, and storage. The architecture from a database used in development to another at production time, that's going to have better handling and faster read against high concurrency, implementing efficient PySpark components of messages and their processing, handling streamed responses during the analysis of models and LLMs output before user feedback by utilizing asynchronous methods where applicable. Hence, scaling considerations across services on GCP using Pulumi IaC reduces provisioning time before runtime upon containerization.
* **Complexity of Deployment** - Although the system will be managed by Docker and Pulumi, the interaction between the different systems brings about complexity, which introduces bugs, hence, CI/CD reduces the risks by deploying and testing automatically.

**1.4 Design Goals and Objectives**

The following are the design goals and objectives of the real-time medical image analysis system which will be a near real-time system with high accuracy and efficiency:

* Achieving real time performance - the system will ensure near real-time analytics outcomes with the LLM responses for efficiency continuity within the user experience since this is of paramount importance in time-critical medical applications. Typical target response time will be less than 3 seconds at uploading an image and running a standard LLM Query.
* Ensuring robust security and privacy – by establishing strict security measures including but not limited to encryption of data both in rest and transit, authentication, and authorization at each section in the system for a given request and input validation during image processing after or during queuing from models via stream at the server end or at API endpoint, hence securely processing the patient data and other users' associated data by using mechanisms or libraries suited for that task. Considering medical regulations, during and after development and the benchmarks to comply with frameworks will also be of importance.
* Scalable Architecture - Designing the system will easily allow horizontal scaling for varied loads or based on the usage from users requesting services. This efficient method handles increased user traffic and volume of data when its adoption increases. Auto-scaling using load balancing and other measures for fault tolerance, such as replication, leverages redundancy while distributing the requests across several servers running as different cloud compute instances, helping to ensure system uptimes are not grossly affected to maintain application responsiveness.
* Intuitive User Interface – the system will provide an intuitive and accessible Streamlit user interface, and prefer the simplest and most straightforward usability. This captures user input through design interaction using forms and questionnaires on specific challenges or any potential issues found in real time and features in use by professionals, especially medical exports. This, in return, improves the overall efficiency of improvement tools in the general workflow and reduces verification errors before recommendations and this eventually leads to a more reliable platform for improving healthcare.

**2. SYSTEM ARCHITECTURE**

Since my project uses a microservice architecture, there is modularity, maintainability, and independent scalability. This distributed architecture permits optimization at the component level with minimum resources for high-throughput requests in a cost-efficient way.

**2.1 System Components**

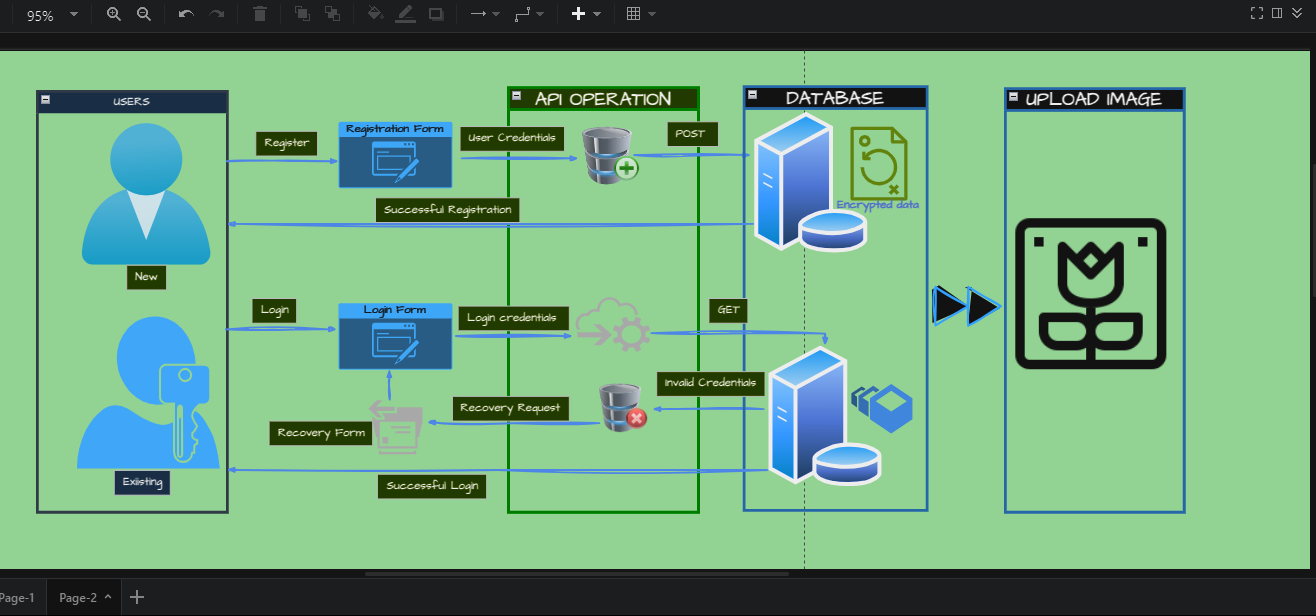
**2.1.1 Frontend UI**

The frontend or client side will be made using Streamlit which provides an intuitive and responsive web interface for uploading images by users for which they see analysis results and also a chatting interface with an explainable AI. Streamlit brings one important declarative approach in developing its UIs. Declarative aspects enable it to develop lighter interfaces, updating efficiently through components and creating value in interface presentation for streamed content after model analysis before presentation. It streams text inputs for LLM queries to handle chats.

**2.1.2 Backend Services**

The system will use Django to handle the backend for user authentication and authorization. This will be a mix-up of custom users and admin users but they will entirely be authenticated using **CSRF** session tokens to ensure data integrity and complex password hashers to make sure that the passwords are fully encrypted in transit and at rest i.e. in database. There will also be an incorporation of a **data pipeline**, which will be running parallel to the backend, will store **RESTful API** endpoints, part of user credential and medical image storage mechanisms for different scenarios like caching, interact with Streamlit at prototype phase and also incorporate the logics that can perform real-time chat, secure API key storage, manage user sessions using the right identifiers at any given time of the analysis process, and upload preprocessed output to a cloud database.

*The following diagram shows how users will be granted access to the system and how the backend logic will handle the different user authorization requests.*

**

**2.1.3 Restful APIs**

The REST API module would be the core interaction module that would bind the entire system. The module will, within itself, will provide certain defined endpoints with which the Streamlit frontend can interact with the backend logic and fetch the results of the analyses. The API also will handle requests from the data pipeline, initiate analysis tasks, and facilitate communication with the Gemini and LLM APIs. It will be responsible for authentication and authorization using Django REST Framework and as such, it will ensure that data serialization is done securely and efficiently enough for scalability in exchanging data among the components of the system. This allows appropriate streaming libraries for models access from user-uploaded medical-related content such as images and chats. Hence, this would be useful to optimize performance, since at times there are numerous sessions handled in real time upon getting concurrent requests by means of various or separated containerized microservices during deployments of either the frontend or backend.

**2.1.4 Gemini API**

The idea of the project still remains the same here i.e. to make several pre-trained models available through an API to perform various medical image tasks, including but not limited to pneumonia detection and fracture detection. Thus, its API is designed to handle high throughput of streamed content hence improving the efficiency in the way it handles the output. I will employ an approach of receiving images and text data streams to different internal specialized models that process them, add error handling and introduce user authentication keys for the API ingestion.

**2.1.5 Large Language Model (LLM) API** for explainable AI chat.

This will be integrated in the system to generate detailed explanations in natural language from the output of Gemini models through streamed queries and will include medical references. It will also address user questions relating to the analysis thus improving reliability. It will also handle and keep track of statistics of usage or request quota for rate limits when implemented during dev-scaling or when testing with some dummy user dataset of real samples from valid image sources, feedback questions close to actual instances, throughput under a limited user during prototype testing and before release.

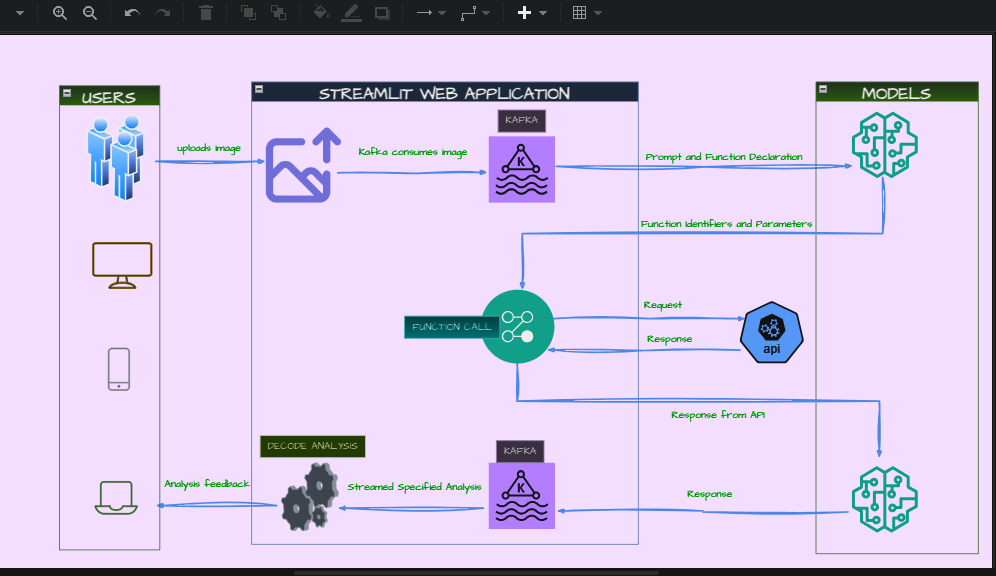
**2.1.6 Data Pipelines**

**2.1.6.1 Apache Kafka**

Kafka offers a really highly scalable, distributed message queue that can enable the interaction of Streamlit, Django, and the Gemini API in an extremely efficient way concerning real-time streaming of big or small data by queuing through optimized topics for increased throughput. Specific Kafka topics are organized through a suitable naming convention in structuring data streams to enable better ingestion by designed schemas. Each of these **consumers** subscribes to an appropriate queue, receives messages from the particular **unique ID**, parses them, and checks the content for validity before routing for specific tasks. The topic will also make use of dedicated topics for error isolation through monitoring of certain messages and queues, hence making validation easy. Therefore, it is easy for resilient mechanisms to detect and probably raise alerts under critical conditions like service failures, which need quick responses to avoid slowing down or breaking the system functionality. The following is how I’ll make use of Kafka:

* **Image Analysis Requests** – Kafka will ingest an uploaded image, ID of the task, and data concerning the session through properly prepared schemas. It will send the user data and keep it for any analysis or job tracking. Then the Messages schema will be designed using standards and patterns where headers and body, and even timestamps from any part in the system that may also be included or their implementation can contain other required fields.
* **Image analysis Report/Response** – after analysis is done by the pre-trained models, Kafka will return the output with appropriate session IDs attached. It also implements the schema design patterns which involve other additional features that include validation flags to confirm the validity of data. Other optional mechanisms that will be included for integrity purposes will be to involve checksum functions for every chunk stream sent or similar implementation when retrieving result after full request or receive using HTTP for streaming files without storage using related libraries since data format can be varied like bytes with or without any encoding or standardized encoding format.
* **User Feedback from Chat** – Kafka will be used to stream data from chat regarding the session and timestamp of the user's response and AI. Other attributes will be provided in any design data structure if required in the future refactoring of this project.

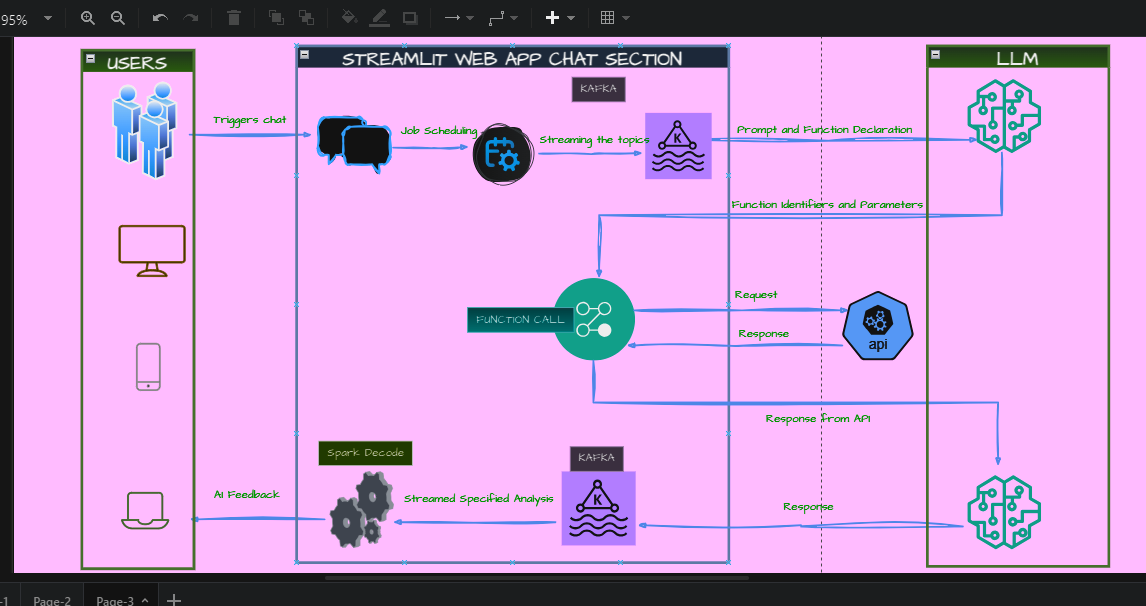
*The following diagram showcases how Kafka will be used to streamline the flow of requests i.e. consumed topics and responses i.e. producer feedback in the system.*

**

**2.1.6.2 Apache Spark**

The code will be designed using PySpark functionality so that it will scale the said pipeline through distributed processing when requests become heavy with various types of analyses. The requests will be preprocessed to extract data required, such as size, channel information, and data format to convert, before passing or checking the models within the clusters using the stream technique after the consumers send and trigger a request call. It will handle the response through streaming of results without permanent persistence once the results are obtained. Furthermore, Spark will provide models for LLM analysis on feedback or text, which can be used in such a manner to produce meaningful outputs in many cases to help interpret and have a track of validating the expected throughput in each and every message sent across and log metrics accordingly at various stages of the pipeline. Also, it will track queues and tasks to know more about failures and recovery to avoid any stream system downtime by isolating the problematic nodes in the system.

The following diagram showcases how Spark will be conjunctively used with Kafka majorly in the chat feature.



**2.1.7 Database Server** (**SQLite3 / GCP Storage**)

**SQLite** can be used in the prototyping period, but for production, Cloud SQL and Cloud Spanner will be put to use. They allow ways to automate the configuration of backups, thus, these provide a secure connection to instances in the cloud. The same may serve useful for retrieving data efficiently, handle their backups during migration from one phase to another i.e. in post-production via various strategies like cloning/dumping an instance, restoring on the cloud from a backup kept locally, or migrating to the cloud via replication for the same database throughout the application in prototyping, testing and production to keep or make use of permanent IDs in a more refined design-make whereby they are match throughout the application under development at the time of checking its data with respect to a particular user e.g., email ID.

**Users Table Structure**

user\_id (INTEGER AUTOINCREMENT)

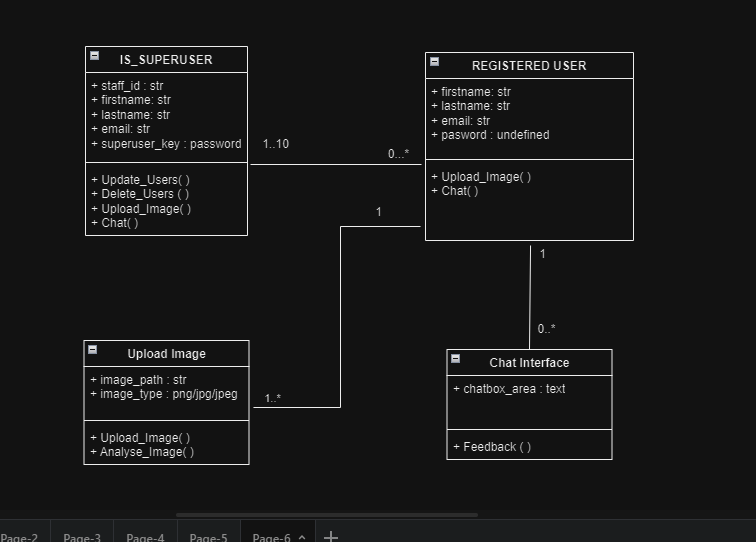
username TEXT **PRIMARY KEY** NOT NULL

email TEXT UNIQUE NOT NULL - may use email as a login and as a unique ID and go without user\_id

password TEXT NOT NULL (store securely by hashing)

image\_path TEXT NULLABLE Path or ID to the user's most recently uploaded image. Defaults to NULL

*The following is a classical class diagram depicting basic methods and attributes for the initial prototype that will be released before deployment.*



***The following tables will be implemented later in production not during the first prototype.***

**ImageAnalysisResults Table**

result\_id INTEGER PRIMARY KEY AUTOINCREMENT

user\_id INTEGER, FOREIGN KEY referencing into Users or email, FK

session\_id for sessions using UUID format. We shall consider using UUIDs as the primary key also.

model\_name TEXT: Name of the particular model used, e.g., pneumonia\_model 1.0

result\_gemini (TEXT) — Raw JSON output from the Model API

analysis\_timestamp TIMESTAMP

**ChatHistory Table**

message\_id (INTEGER PRIMARY KEY AUTOINCREMENT)

result\_id INTEGER FOREIGN KEY referencing ImageAnalysisResults or even better session\_id as Foreign Key.

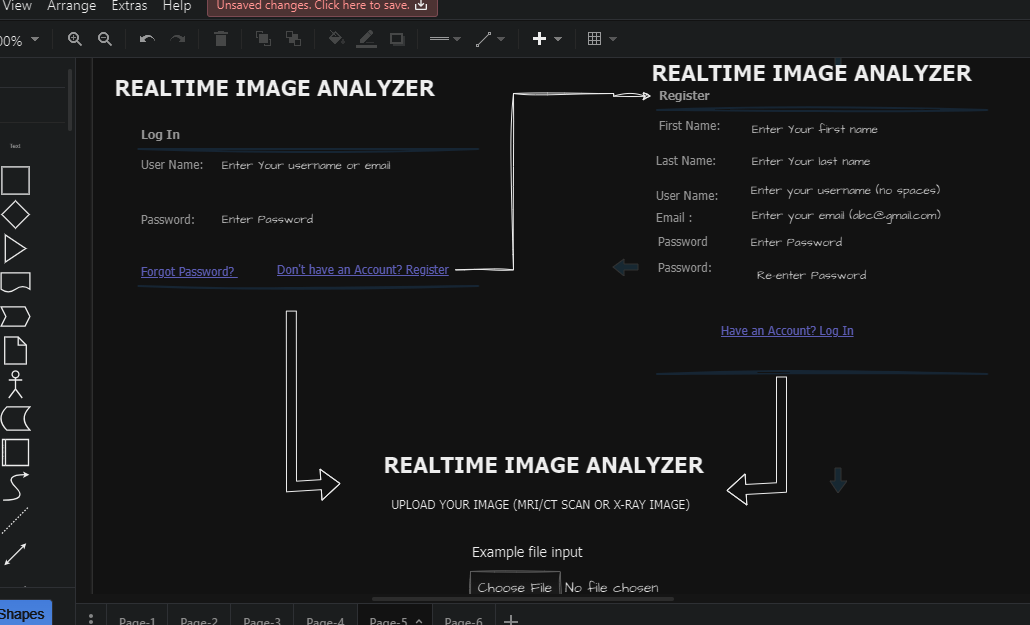
user\_message (TEXT)

message\_timestamp (TIMESTAMP)

**3.0 USER INTERFACE DESIGN**

**3.1 Frontend User Interface**

**3.1.1 Registration/Login Pages –** The system will have a nice and simple registration and login page for new and existing users respectively. The following diagram showcases the flow of registration and login when users enter their credentials to the system.

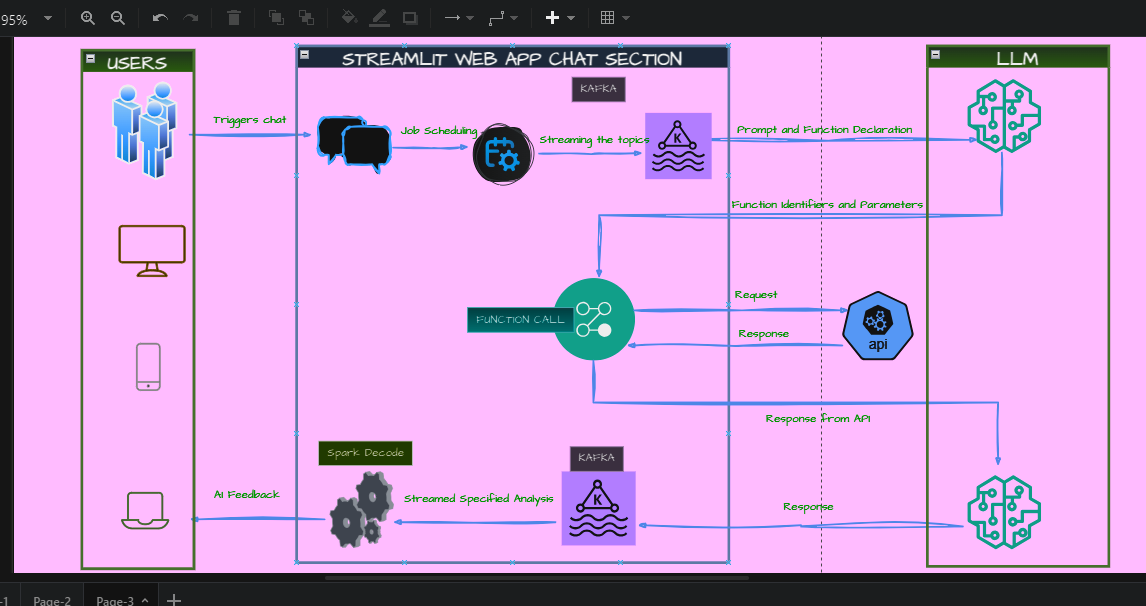


**3.1.2 Uploading an Image** - Streamlit’ s frontend will support the following method of uploading image files in the system:

* Newly uploaded image data will override what might be cached locally. On completion of prototyping, especially for the persistence of image storing, this gets updated with unique path information relevant for the given uploaded image by a given user and after confirmation that image saves into the database were successful.
* One of the efficient mechanisms which will be implemented is to involve checksums at pre and post uploads, with the use of suitable secure libraries before being stored permanently. Resource limits will also be set through appropriate size constraints of memory use.

**3.1.3 Analysis Results Display** - The module shall provide intuitive visualizations for model analysis results. Depending on the type of analysis, the chart will depict specific key information or important highlights of the analysis. Examples include confidence score visualizations from different tests, enabling enhanced overall quality and accuracy in verification for correctness by medical experts specifically when being used for a final assessment in treatment since it is not completely dependent on system-generated output as medical field is all about probabilities.

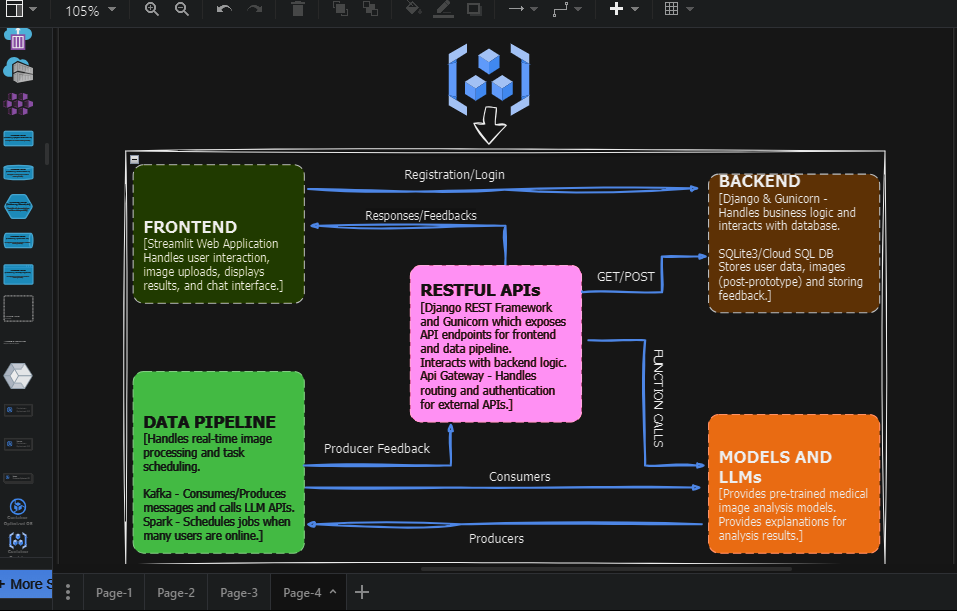
**3.1.4 Active Chat Interface** - Streamlit, with its respective libraries, features an interactive UI, keeping in mind optimized caching while streaming requests like text. And for this case, huge requests will not be sent out because they block the UI and enough data will be sent in batches to the models to avoid huge messages in a queue from filling up or exceeding our set Memory Quota. The following diagram showcases the intended flow of interactions with the Chat feature.



**4.0 DEPLOYMENT** using Docker and Pulumi IaaC

**4.1 Containerization using Docker**

In **docker**, the components will be running as isolated containers i.e. light-weight images which speed up instances starting in the case of scaling by auto-scaling mechanisms and in the event of an instance failure when monitored.The following diagram showcases the anticipated deployment containerization architecture of the system components.



**4.2 Infrastructure Management using Pulumi**

For the case of **Pulumi IaaC**, all associated infrastructure will then be managed during and post-deployment of various services so that upgrades could be handled through configurations in the environment and directly deployed with ease to the cloud.

**5.0 SECURITY CONSIDERATIONS**

**5.0.1 API Security** – This project will employ secure, cloud-based encrypted API key management and controlled access since updating or perusal of keys, which will be involved quite regularly within the production environment, will ensure that the system is well secured concerning API security terms outlined in the following Data Security topic.

* **Data Security** for data at Rest & in-Transit - Since HTTPS usage, during a call to models, happens at the http layers, this inhibits malicious access/sniffers from stealing users' medical or sensitive-related image data at the instance it's on the streaming pipeline within an API call containing image byte stream data. User requests will be dealt with accordingly, using suitable serializers from any libraries ,for my case I’ll use Django serializers, so that all messaging systems would look standardized across each and every process or call associated. Furthermore, to avoid malicious script insertions through different ways, I’ll handle file readings by providing a temporary random session path, or by bringing in cloud isolated file handlers which create their encrypted and isolated sessions linked with unique hashes, including data of their original paths before any opening of files on the server side. This will be another extra but more secure step rather than taking a risk from those vulnerabilities. It will be able to let the user authenticate and authorize using tokens for session identification if implemented with email for identifiers after verification of the sender and should implement a simple Role-based access control (RBAC) model on particular sensitive features within the app, checking before handling the request by giving each component a certain level of security through well-established or secure programming standards, styles, and using appropriate and actively patched, secure and maintained libraries following better practices laid out through the newest forms of compliance certification in Data Acts.

**5.0.2 Scalability and Enhancements -** **Horizontal Scaling with Load Balancing**

The project will majorly scale horizontally by replicating into multiple geographic zones and it could easily reroute requests and manage a large number of users and make it highly available with reduced latency by implementing proper or, rather efficient automated scaling configurations on servers during deployment with the help of other services on the cloud. The result should possess increased fault tolerance and be easier to restore in case of crashes of any systems without disturbing the performance in real time by assuring speedier system boots because of the use of simplistic but superior, actively maintained code libraries.

**5.0.3 Distributed Database -** During transition in future deployments after this project’s prototype for scalable schema design during data management, this kind of distribution will enable seamless transition and analytically perform auto-update to point application and data pipeline services correctly with the use of suitable configuration. This will probably be done by enabling cloud database by dumping and recreating it with the triggers for correct migration, centralized logging and error aggregation to enable better system monitoring across disparate asynchronous processes serving to realize effective identification and resolution in complex, concurrent real-time systems across different request loads-so that throughput will be known at each layer throughout the life of the RMIA system.

**6.0 Appendix**

The following are some abbreviations that I have used in the document

**Abbreviations**

RMIA – Realtime Medical Image Analyzer

IaC/IaaC – Infrastructure As Code

HTTPS – Hypertext Transfer Protocol Secure

API – Application Programming Interface

LLM – Large Language Models

UI – User Interface

CI/CD – Continuous Integration/Continuous Delivery

GCP – Google Cloud Platform

AI -Artificial Intelligence

CSRF – Cross-Site Request Forgery