Software Design Document

**Project Name**: ClinIQ – GenAI-Powered Lab Report Analyzer  
**Author**: Yogya C  
**Date**: April 2025  
**Version**: 1.0

### 1. Introduction

ClinIQ is a GenAI-powered web application designed to analyze and summarize medical lab reports (e.g., hematology and urinalysis) by leveraging OCR, a local LLM, and a vector database. The system extracts values, retrieves relevant reference knowledge, and generates medically contextualized summaries and insights.

### 2. Document Outline

This design document outlines the full system design for ClinIQ, including its architectural decisions, constraints, and development strategies. It walks through each component's design, rationale, and implementation-level decisions for scalability and maintainability.

### 3. Document Description

## 3.1 Introduction

This software design document provides the architectural and detailed design of the ClinIQ platform. It aims to ensure the system is modular, production-grade, and scalable, particularly under high concurrency.

## 3.2 System Overview

ClinIQ ingests lab reports as PDFs, performs OCR using Tesseract, and uses embeddings to retrieve related medical facts from a reference corpus stored in a vector database. The core GenAI engine (LLM via Ollama) combines OCR output and reference knowledge to generate interpretations. The system supports multiple concurrent users and asynchronous task handling.

## 3.3 Design Considerations

The system is optimized for asynchronous workflows, offline capability, and LLM inference performance. Scalability, fault tolerance, and explainability are prioritized.

## 3.3.1 Assumptions and Dependencies

* Users upload well-formatted PDF lab reports
* Tesseract OCR performs reliably on provided input
* Ollama model is available locally via API
* Vector DB (initially Chroma, later Pinecone or Qdrant)
* Celery and Redis for task queueing

## 3.3.2 General Constraints

* Local-only LLM (no external API calls)
* Max concurrent user goal: 300+
* Vector DB must scale to 100k+ documents
* Response time must stay <5 seconds for 90% of requests

## 3.3.3 Goals and Guidelines

* Modular microservice-friendly architecture
* Use of FastAPI for async support and performance
* All long-running operations handled as background tasks
* Replace polling with event-driven responses

## 3.3.4 Development Methods

* GitHub Actions for CI/CD
* Unit + integration testing
* Containerized via Docker for portability

## 3.4 Architectural Strategies

* Adopt FastAPI + Celery for request/task separation
* Switch from polling to SSE/WebSockets or Redis pub-sub
* Introduce semantic similarity scoring thresholding for vector retrieval
* Split documents into semantic chunks using RecursiveCharacterTextSplitter

## 3.5 System Architecture

* **Frontend**: HTML/JS UI with file upload, summary view, and progress updates via SSE
* **Backend (FastAPI)**: Handles requests, queues tasks, and serves LLM responses
* **OCR Worker**: Async Celery task using Tesseract to extract text
* **Vector DB**: Stores and retrieves reference facts (Pinecone or Qdrant)
* **LLM Inference**: Uses local Ollama to generate summary
* **State Store**: Redis for tracking job status

## 3.5.1 Subsystem Architecture

* **UploadHandler**: Accepts and saves files, starts OCR task
* **OCRProcessor**: Runs OCR and extracts fields
* **EmbedAndRetrieve**: Embeds text and queries vector DB
* **LLMResponder**: Forms prompt and calls Ollama
* **Notifier**: Publishes completion events to client via SSE/WebSocket

## 3.6 Policies and Tactics

* Log every step (ingestion, chunking, retrieval, generation)
* Return similarity score for top retrieval
* Retry on LLM inference failure (with exponential backoff)
* Cap LLM runtime and vector search timeouts
* Enable autoscaling and concurrent workers (via Gunicorn + Uvicorn + Celery)

## 3.7 Detailed System Design

UploadHandler

* *Responsibilities*: Handle uploads, validate files, enqueue OCR job
* *Constraints*: PDF format only
* *Composition*: FastAPI route + Redis enqueue logic
* *Uses*: upload\_file, enqueue\_ocr\_task
* Interface: /upload endpoint

OCRProcessor

* *Responsibilities*: OCR + text cleaning
* *Constraints*: OCR time <3s per page
* *Composition*: Celery task + Tesseract engine
* *Uses*: Tesseract, PDF parser
* *Exports*: Extracted text + metadata

EmbedAndRetrieve

* *Responsibilities*: Chunk text, embed, query vector DB
* *Constraints*: Top-k results in <500ms
* *Composition*: LangChain + Pinecone
* *Uses*: Recursive splitter + embedding model
* *Exports*: Matching documents + similarity score

LLMResponder

* *Responsibilities*: Prompt + inference using Ollama
* *Constraints*: Prompt size <2k tokens, timeout 10s
* *Composition*: Ollama API call + templated prompt
* *Uses*: OCR result + vector context
* *Exports*: Final summary

Notifier

* *Responsibilities*: Notify frontend when task is complete
* *Constraints*: Use SSE/WebSocket
* *Composition*: Redis pub-sub + FastAPI event stream
* *Uses*: Redis keywatcher or push system
* *Exports*: Push status: complete and result

### 4. Glossary

* **OCR**: Optical Character Recognition
* **LLM**: Large Language Model
* **SSE**: Server-Sent Events
* **RAG**: Retrieval-Augmented Generation
* ChromaDB/Pinecone: Vector databases
* **Celery**: Asynchronous task manager

### 5. Bibliography

* LangChain Documentation
* Pinecone Documentation
* Ollama Model Reference
* [Tesseract OCR Project](https://github.com/tesseract-ocr/tesseract)
* FastAPI Docs
* Redis PubSub Pattern
* LangChain Chunking Strategies