**PROJECT REPORT**

**TABLE OF CONTENTS**

| **Chapter No.** | **Title** | **Page No.** |
| --- | --- | --- |
| - | **ABSTRACT** | iv |
| - | **LIST OF FIGURES** | vii |
| **I** | **INTRODUCTION** | 1 |
| 1.1 | Domain Introduction | 1 |
| 1.2 | Objective | 2 |
| 1.3 | Scope of the Project | 2 |
| **II** | **LITERATURE REVIEW** | 3 |
| **2.1** | Comparison With Existing Systems | 3 |
| **III** | **SYSTEM ANALYTICS** | 5 |
| 3.1 | Existing Problem | 5 |
| 3.1.1 | Disadvantages of Existing System | 5 |
| 3.1.2 | Advantages of Existing System | 6 |
| 3.2 | Proposed Methodology | 6 |
| 3.2.1 | Advantages of Proposed System | 7 |
| **IV** | **SYSTEM REQUIREMENTS** | 8 |
| 4.1 | Hardware Requirements | 8 |
| 4.2 | Software Requirements | 8 |
| 4.3 | Required Python Libraries | 9 |
| 4.4 | Programming Languages & Frameworks Used | 9 |
| **V** | **MODELS AND METHODS** | 10 |
| 5.1 | Module Overview | 10 |
| 5.2 | Regression Analysis for Symptom Prediction | 11 |
| 5.3 | LSTM for Pattern Learning | 11 |
| 5.4 | Retrieval-Augmented Generation (RAG) Chatbot | 12 |
| **VI** | **IMAGE-BASED DIAGNOSIS MODULE** | 13 |
| 6.1 | Role of Image Processing | 13 |
| 6.2 | GROQ API Integration | 13 |
| 6.3 | Prompt Engineering with Images | 14 |
| 6.4 | Sample Input/Output Examples | 15 |
| **VII** | **IMPLEMENTATION** | 16 |
| 7.1 | Input Processing Flow | 16 |
| 7.2 | Integration Between Frontend, Backend, and AI | 17 |
| 7.3 | MongoDB Vector Search with Session Chat | 18 |
| **VIII** | **SYSTEM DESIGN** | 19 |
| 8.1 | System Architecture Diagram | 19 |
| 8.2 | Use Case Diagram | 20 |
| 8.3 | Class Diagram | 21 |
| 8.4 | Activity Diagram | 22 |
| 8.5 | Flowchart | 23 |
| **IX** | **MODULE DESCRIPTION** | 24 |
| 9.1 | Data Collection | 24 |
| 9.2 | Data Preprocessing | 25 |
| 9.3 | Feature Extraction | 25 |
| 9.4 | LSTM Training | 26 |
| 9.5 | Prediction and Diagnosis | 26 |
| **X** | **SYSTEM TESTING** | 27 |
| 10.1 | Testing Strategy | 27 |
| 10.2 | Types of Testing (Unit, Integration, System) | 28 |
| **XI** | **CONCLUSION AND FUTURE WORK** | 29 |
| - | **GLOSSARY** (RAG, LSTM, Vector DB, etc.) | 30 |
| - | **APPENDIX A** – Sample Dataset / Image Input | 31 |
| - | **APPENDIX B** – Screenshots of Chatbot UI & Image Module | 32 |
| - | **APPENDIX C** – **User Guide** (How to use chatbot & image tool) | 33 |
| - | **REFERENCES** | 34 |

**ABSTRACT**

In today’s fast-paced world, timely and accurate medical guidance is crucial. Many people turn to the internet for health-related queries, which often leads to confusion or misdiagnosis. To address this issue, we present an AI-powered **Symptom-Based Medical Diagnosis Chatbot** that uses **Retrieval-Augmented Generation (RAG)** and **Machine Learning (ML)** to provide early-stage medical insights based on user-described symptoms.

The chatbot accepts text inputs from users and predicts possible diseases using trained models like **Linear Regression** and **LSTM (Long Short-Term Memory)**. It also uses **RAG architecture** to enhance answer accuracy by retrieving relevant data from a pre-built medical knowledge base, improving both context and response relevance.

An additional feature of the system is the **Image-Based Diagnosis Module**. Users can upload images (e.g., skin rashes), which are processed using the **GROQ AI API** to generate a response that considers both visual and textual symptoms. This multi-modal analysis improves diagnostic quality and mimics real-world consultation.

The system is built using a **FastAPI backend** and a **MERN stack frontend**, with session-based chat history stored in **MongoDB**. The chatbot provides quick, relevant, and responsible replies, and suggests professional consultation for serious conditions.

This project aims to make preliminary healthcare more accessible, especially in remote areas. It is not a replacement for doctors but a helpful tool for early symptom analysis and awareness.

**CHAPTER 1 – INTRODUCTION**

**1.1 Domain Introduction**

Healthcare has always been one of the most critical sectors in human society. With the increasing global population and the emergence of new diseases, healthcare systems are under constant pressure to deliver accurate diagnoses, provide personalized care, and ensure timely treatments. Traditional methods of medical diagnosis rely heavily on physical consultations and manual evaluation by healthcare professionals, which can be time-consuming, expensive, and inaccessible in remote or underserved areas.

In this digital era, **Artificial Intelligence (AI)** has proven to be a transformative force in the medical domain. From image-based diagnostics to predictive analytics and patient monitoring, AI is increasingly being integrated into healthcare systems. Among the various AI applications, **AI-based medical chatbots** have emerged as a game-changer, offering round-the-clock health advice, symptom assessment, and early-stage diagnosis through user interaction.

The domain of this project specifically lies in the **intersection of Natural Language Processing (NLP), Machine Learning (ML), and Retrieval-Augmented Generation (RAG)**. Our system leverages these technologies to develop an intelligent medical chatbot capable of analyzing user-reported symptoms, querying relevant medical information, and generating accurate, human-like responses. The chatbot also supports **image-based diagnosis** using visual symptom analysis powered by the **GROQ AI API**, adding an additional layer of functionality.

This project is designed not to replace doctors, but to act as an **assistive diagnostic tool**—especially helpful in triaging symptoms, providing preventive advice, and promoting health awareness, all while reducing the load on the healthcare infrastructure.

**1.2 Objective**

The primary goal of this project is to develop an **AI-powered medical diagnosis chatbot** that can interact with users in natural language, accept symptoms or medical queries, and return a probable diagnosis or health guidance. The system combines **machine learning prediction models**, **LSTM-based sequence learning**, **RAG for context-aware responses**, and **image understanding via the GROQ API** to deliver a comprehensive diagnosis assistant.

**Specific Objectives:**

* ✅ To create a **text-based medical chatbot** capable of understanding and processing natural language symptom descriptions.
* ✅ To use **ML regression and LSTM models** to identify possible diseases based on symptom patterns.
* ✅ To integrate a **RAG-based NLP pipeline** that enhances chatbot intelligence by referencing vector-based medical data sources.
* ✅ To allow users to upload images (e.g., rashes, swelling, eye conditions) and receive an image-based diagnostic response using **GROQ AI**.
* ✅ To provide accurate and informative responses while also flagging “unknown symptoms” to maintain system integrity.
* ✅ To maintain user session history and secure communication using a **MERN stack** and **FastAPI backend**.

In summary, the objective is to bridge the gap between users and early medical insights through intelligent automation that combines multiple AI modalities.

**1.3 Scope of the Project**

This project is designed to address the growing need for **smart, accessible, and accurate symptom analysis tools** that can support users anytime and anywhere. The scope of this system is wide-ranging and addresses several aspects of medical assistance using AI technologies.

**Key Aspects of the Project Scope:**

* **Symptom-to-Disease Analysis**:  
  The chatbot takes natural language input describing symptoms and processes it using trained ML models (including regression and LSTM) to predict potential diseases.
* **Retrieval-Augmented Responses**:  
  Instead of relying solely on static answers, the system retrieves relevant documents and vectorized data to generate dynamic, **context-aware replies** using the RAG model.
* **Image-Based Diagnosis**:  
  Users can submit images alongside text queries. The system uses **GROQ’s LLaMA 4-based Maverick model** to analyze the image in conjunction with the query and return meaningful diagnostic information.
* **Conversational UI with History Support**:  
  The chatbot mimics a human-like interaction model, storing conversation history using MongoDB and rendering it in a **ChatGPT-style UI**. It maintains user session context for better personalization.
* **Platform Independence**:  
  Built using a combination of **FastAPI (backend)** and **MERN stack (frontend)**, the application is scalable and can be hosted across multiple platforms, including cloud services.
* **Target Users**:
  + Individuals seeking preliminary health advice.
  + Patients in remote areas without immediate access to doctors.
  + Healthcare startups or institutions looking to integrate AI solutions.
  + Researchers and developers studying AI-based diagnosis tools.

This project is not meant to provide medical prescriptions but to **act as a pre-diagnosis assistant** and recommend users to consult medical professionals when necessary.

**CHAPTER II. LITERATURE REVIEW**

**2.1 Comparison with Existing Systems**

Medical diagnosis chatbots and symptom checkers have gained popularity as preliminary tools to assist users in understanding possible health conditions. Traditional symptom-based systems often rely on static rule-based algorithms or decision trees, which match user symptoms against predefined symptom-disease mappings. These systems usually require users to enter exact symptom names, limiting their flexibility and accuracy. For example, many existing systems fail to handle variations in symptom descriptions or incomplete data, leading to incorrect or vague diagnoses.

In contrast, recent advances utilize machine learning (ML) models such as Logistic Regression, Random Forest, and LSTM networks to capture complex patterns from medical datasets. These models can generalize better to unseen or partially described symptoms. However, standalone ML models often lack sufficient contextual understanding and depend heavily on the quality and coverage of the training data.

The proposed system combines **Retrieval-Augmented Generation (RAG)** with ML, integrating a knowledge retrieval system that fetches relevant medical documents or data snippets before generating an answer. This hybrid approach improves response accuracy, especially for ambiguous or complex queries, by grounding the chatbot’s reply in verified medical knowledge. Moreover, adding image-based diagnosis using GROQ AI further distinguishes the system by enabling multimodal input analysis, which is rare among current solutions.

Compared to purely text-based or rule-based systems, the hybrid RAG-ML chatbot offers more robust, flexible, and contextually relevant diagnoses, enhancing user trust and utility.

**2.2 Performance Metrics in Existing Systems**

Evaluating medical diagnosis chatbots involves several performance metrics that measure prediction accuracy, relevance, and user satisfaction. Common metrics used in literature include:

* **Accuracy**: The proportion of correct disease predictions out of total cases. High accuracy indicates reliable symptom-to-disease mapping but can be misleading if data is imbalanced.
* **Precision and Recall**: Precision measures how many predicted diseases were actually correct, while recall measures how many true diseases were correctly identified. Balancing these metrics is crucial to avoid false positives (wrong diagnoses) and false negatives (missed diagnoses).
* **F1 Score**: Harmonic mean of precision and recall, offering a single metric balancing both.
* **Response Time**: The time the chatbot takes to generate an answer, impacting user experience.
* **User Satisfaction**: Often measured through surveys or qualitative feedback, assessing how useful or trustworthy users find the chatbot.

In existing ML-based chatbots, accuracy typically ranges from 70% to 85% depending on dataset size and complexity. RAG-enhanced systems show improvements in providing contextually accurate answers, especially in ambiguous cases, reducing irrelevant or hallucinated responses.

Image-based diagnosis modules are evaluated based on classification accuracy or image recognition precision, but integrating this with symptom-based text input remains a developing area.

Our project focuses on achieving high accuracy and contextual relevance by combining ML prediction models with RAG, while also incorporating image analysis for enhanced diagnostics.

**Image Suggestions:**

* **Diagram comparing traditional rule-based vs ML-based vs RAG-based chatbot workflows** (for 2.1)
* **Table showing typical performance metrics (accuracy, precision, recall) of existing symptom checkers** (for 2.2)
* **Flowchart of multimodal diagnosis combining text and image inputs** (optional)

**III. SYSTEM ANALYTICS**

**3.1 Existing Problem**

Medical diagnosis through automated systems faces several challenges. Existing symptom-based checkers primarily rely on exact keyword matching or rule-based approaches, requiring users to input symptoms in a very specific manner. This rigidity often leads to misdiagnosis or an inability to provide meaningful results when the symptom description deviates from predefined terms. Moreover, many systems are limited to textual input and do not support multimodal data like images, which can provide crucial diagnostic information (e.g., skin rashes, swelling, or abnormal growths).

These systems also lack context awareness and the ability to explain or justify their predictions in human-readable terms, which can reduce user trust. Additionally, they often fail to update dynamically with new medical knowledge or research findings, making them less reliable over time.

**3.1.1 Disadvantages of Existing System**

* **Strict symptom input requirements:** Users must provide symptoms exactly as expected by the system, limiting usability.
* **Lack of contextual understanding:** Systems do not interpret symptom combinations holistically, leading to inaccurate or incomplete diagnosis.
* **Inability to handle vague or complex queries:** The chatbot can’t effectively process partial or ambiguous symptoms.
* **No multimodal input:** Most systems cannot analyze images or other media, missing vital diagnostic cues.
* **Static knowledge base:** Existing systems do not dynamically retrieve updated medical data, risking outdated advice.
* **Limited interaction:** They offer one-way information without interactive follow-ups or clarifications.
* **Low user trust:** Due to limited explanations and frequent incorrect predictions, users may distrust automated systems.

**3.1.2 Advantages of Existing System**

Despite their shortcomings, traditional systems provide benefits:

* **Simplicity:** Rule-based systems are straightforward and easy to implement.
* **Speed:** Quick responses due to direct symptom-to-disease mapping.
* **Baseline guidance:** Useful for preliminary symptom checks before consulting a doctor.
* **Structured outputs:** Often provide clear, concise lists of potential conditions.
* **No complex infrastructure required:** Can run on basic servers without heavy computational resources.

**3.2 Proposed Methodology**

Our project introduces a hybrid methodology combining **Retrieval-Augmented Generation (RAG)** and **Machine Learning (ML)** to overcome existing challenges. The RAG component fetches relevant medical documents or data snippets from a vector database based on user queries, grounding answers in authoritative sources and reducing hallucination.

The ML model, fine-tuned on medical symptom datasets, processes the user’s input (including non-exact symptom descriptions) to predict possible diseases. We also incorporate an **image-based diagnosis module** that analyzes medical images uploaded by users for a more comprehensive evaluation.

This integrated approach enables:

* Flexible symptom input handling with natural language processing.
* Context-aware and explainable responses.
* Dynamic knowledge retrieval ensuring up-to-date information.
* Multimodal diagnosis combining text and images.
* Interactive conversation with clarifications and follow-up questions.

**3.2.1 Advantages of Proposed System**

* **Higher accuracy and robustness:** Combining ML with RAG enables better handling of ambiguous or incomplete symptoms.
* **Improved user experience:** Interactive chatbot interface supports natural conversations and follow-ups.
* **Multimodal inputs:** Ability to analyze images alongside text increases diagnostic coverage.
* **Explainable AI:** Answers are backed by retrieved medical context, improving transparency and trust.
* **Dynamic knowledge base:** Continuous indexing of medical literature keeps the system current.
* **Scalable and adaptable:** New medical data or symptoms can be added without retraining the entire model.
* **Supports preliminary diagnosis:** Helps users understand possible conditions before seeking professional care.

**Suggested Images for This Chapter:**

* **Flowchart or diagram comparing existing system workflow vs proposed RAG + ML system**
* **Table summarizing disadvantages and advantages of existing systems side by side**
* **Architecture diagram of proposed system showing multimodal input and RAG integration**

**IV. SYSTEM REQUIREMENTS**

**4.1 Hardware Requirements**

For your AI-powered medical chatbot system, the hardware must support multiple tasks like natural language processing, vector search, image-based diagnosis, and web service hosting. Here's what you need:

* **Processor (CPU):** A modern multi-core CPU (Intel i5/Ryzen 5 or better) to efficiently handle backend services, including data processing, embedding calculations, and API calls. High concurrency support is beneficial to manage simultaneous user queries.
* **Memory (RAM):** Minimum 8 GB RAM. The chatbot’s backend loads machine learning models (like LSTM and RAG models) and document embeddings, which are memory-intensive. 16 GB is recommended for smoother performance, especially when running local embedding generation or fine-tuning.
* **Storage:** At least 100 GB SSD to store medical PDFs, vector embeddings in MongoDB, model checkpoints, and logs. Fast storage ensures quick access during retrieval and indexing.
* **GPU (Optional but beneficial):** A CUDA-enabled GPU (e.g., NVIDIA GTX 1660 or better) can speed up embedding generation and LSTM inference when using local ML models. However, if you're using cloud-based APIs like GROQ for image diagnosis, GPU is not mandatory.
* **Network:** Stable high-speed internet connection for API calls to GROQ AI image diagnosis and other external services, as well as frontend-backend communication.

This setup ensures your system handles real-time symptom inputs, complex vector similarity searches, and image uploads without lag or downtime.

**4.2 Software Requirements**

The software environment is designed for seamless integration of various components:

* **Operating System:** Linux (Ubuntu 20.04 or higher preferred for server deployment), Windows 10/11 for development and testing, or macOS. Linux is recommended for stability and better resource management on production servers.
* **Database System:** MongoDB (preferably MongoDB Atlas for managed cloud hosting) to store vector embeddings, medical documents metadata, and user chat history efficiently. Vector search capabilities allow for fast similarity queries.
* **Backend Frameworks:**
  + **Python 3.8+** for AI model orchestration, data processing, and running FastAPI microservices for image-based diagnosis.
  + **Node.js 14+** for backend API server that handles frontend requests, routes user inputs to AI services, manages user authentication, and sessions.
* **Frontend Framework:** React.js to build a responsive, user-friendly web interface where users input symptoms and images, and receive diagnoses and recommendations.
* **API Testing Tools:** Postman or Curl for testing REST API endpoints during development.
* **Version Control:** Git for code versioning and collaboration.

These software components are essential for building, running, and scaling the chatbot system while maintaining modularity.

**4.3 Required Python Libraries**

The Python backend relies on specialized libraries for embedding generation, document processing, vector search, and API integration:

* **langchain:** Core framework for building retrieval-augmented generation (RAG) workflows by chaining document retrieval with large language models. It supports integration with MongoDB vector search and embedding models.
* **pymongo:** Official MongoDB driver to connect and perform CRUD operations on MongoDB collections storing vector data and chat logs.
* **pdfplumber:** Extracts text data from medical PDFs for indexing into MongoDB, enabling knowledge retrieval.
* **transformers & sentence-transformers:** Provides pre-trained embedding models like all-mpnet-base-v2 to convert symptom texts and documents into dense vector representations.
* **fastapi:** High-performance web framework used to build the image-based diagnosis microservice, allowing asynchronous request handling and quick responses.
* **requests:** To communicate with external APIs such as GROQ AI for image analysis and diagnosis.
* **python-dotenv:** To securely manage API keys, database credentials, and other sensitive configuration details from .env files.
* **Pillow (PIL):** For image validation and processing before sending images to the diagnosis service.
* **uvicorn:** ASGI server to run FastAPI applications efficiently.
* **re (regular expressions):** Used in cleaning and formatting chatbot responses.

These libraries ensure smooth data flow, ML model interaction, and backend API responsiveness.

**4.4 Programming Languages & Frameworks Used**

Your project is built using technologies chosen for their synergy in AI, web development, and data management:

* **Python:** The backbone language for AI, ML model handling (LSTM, RAG), data preprocessing, PDF text extraction, and the FastAPI microservice for image-based diagnosis.
* **Node.js:** Backend server environment running Express.js or similar framework to handle HTTP requests from the React frontend. It routes symptom inputs and images to Python services and returns responses efficiently.
* **React.js:** Frontend framework responsible for dynamic user interface elements such as symptom input fields, chat interface, image upload, and displaying diagnosis results.
* **MongoDB:** NoSQL database for scalable storage of vector embeddings (for similarity search), medical documents, and user interaction history, enabling fast retrieval.
* **FastAPI:** Used to develop the image diagnosis microservice that accepts image uploads, processes them using GROQ API, and returns medical insights.
* **REST API architecture:** Facilitates communication between frontend, backend, Python AI services, and third-party APIs, maintaining modularity and separation of concerns.

**Suggested Images for This Chapter**

* **System Hardware Setup Diagram:** Showing CPU, RAM, optional GPU, storage, and network components.
* **Software Stack Diagram:** Illustrating OS, Python backend (with ML libraries), Node.js backend, MongoDB, React frontend, and external APIs like GROQ.
* **Library Interaction Flow:** How Python libraries like langchain, pymongo, pdfplumber, and FastAPI interact in the system.

**V. MODELS AND METHODS**

**5.1 Module Overview**

The project integrates several AI and ML techniques to deliver an accurate, interactive medical diagnosis chatbot. The core modules include:

* **Regression Analysis:** A statistical method used for initial symptom-to-disease correlation analysis, helping to predict probable illnesses from numerical symptom data.
* **Long Short-Term Memory (LSTM) Networks:** A type of recurrent neural network (RNN) that excels at learning from sequential data, applied here to detect complex temporal patterns in symptom progressions.
* **Retrieval-Augmented Generation (RAG):** A cutting-edge NLP method combining retrieval of relevant medical knowledge documents with generative language models, enabling the chatbot to provide contextually accurate and detailed responses.
* **Image-Based Diagnosis Module:** (Covered separately) utilizes GROQ AI for analyzing uploaded medical images for additional diagnostic support.

These modules work synergistically: regression and LSTM handle structured symptom data and temporal patterns, while RAG enhances user interaction with rich, knowledge-grounded answers.

**5.2 Regression Analysis for Symptom Prediction**

Regression analysis is used to establish relationships between input symptoms and possible diseases. In this project, multiple regression techniques were explored, including linear and logistic regression, to model how symptom severity scores or presence indicators correlate with diagnoses.

* **Purpose:** Quickly identify statistical correlations that suggest a disease when symptoms are numerically encoded or binarized (present/absent).
* **Data Input:** Patient symptom vectors, e.g., fever=1, cough=0, abdominal pain=1.
* **Output:** Probability scores for various diseases.
* **Strengths:** Provides interpretable coefficients showing symptom importance, useful for preliminary prediction and understanding feature impact.
* **Limitations:** Regression assumes linear or simple relationships, which might not capture complex symptom interactions.

The regression model serves as an initial filter in the chatbot pipeline, flagging likely conditions and narrowing the scope for more nuanced analysis by LSTM and RAG models.

**5.3 LSTM for Pattern Learning**

Long Short-Term Memory (LSTM) networks are specialized for processing sequential data with temporal dependencies — crucial for medical symptoms that evolve over time.

* **Why LSTM?** Symptoms often do not appear all at once; their progression and order matter. For example, early fatigue followed by jaundice might indicate a different diagnosis than jaundice appearing first.
* **Input Format:** Sequences of symptom occurrences over time. Each symptom is encoded in a time-step vector.
* **Architecture Details:**
  + LSTM layers with forget, input, and output gates manage information flow, preventing the vanishing gradient problem common in RNNs.
  + The model learns to recognize temporal symptom patterns linked to specific diseases.
* **Output:** Predicted disease class or probability distribution over diseases.
* **Training Data:** Historical symptom-disease datasets or synthetically generated sequences based on medical knowledge.
* **Benefits:** Captures non-linear, complex dependencies over time that simple regression misses, improving prediction accuracy for diseases with evolving symptom profiles.

LSTM thus complements regression by modeling the dynamic nature of symptom presentation, enhancing chatbot prediction reliability.

**5.4 Retrieval-Augmented Generation (RAG) Chatbot**

The heart of the system is the Retrieval-Augmented Generation (RAG) architecture, which combines knowledge retrieval and natural language generation for intelligent, context-aware conversation.

* **Core Idea:** Instead of relying solely on a fixed language model’s training, RAG dynamically fetches relevant documents from a vector database of medical literature and uses that knowledge to generate responses.
* **Components:**
  + **Retriever:** Uses vector embeddings (from models like all-mpnet-base-v2) to find top-k relevant documents or passages based on user symptom queries.
  + **Generator:** A large language model (e.g., meta-llama/llama-4-maverick) that produces human-like, informative responses conditioned on the retrieved context.
* **Workflow:**
  + User inputs symptoms.
  + Symptoms converted to embeddings.
  + Retriever queries MongoDB vector store for relevant medical text.
  + Generator combines retrieved text with user input to formulate precise answers.
* **Advantages:**
  + Up-to-date and context-specific answers grounded in verified medical texts.
  + Avoids hallucinations common in pure generation models by grounding output in real documents.
  + Can explain reasoning, list tests, and recommend further action, improving user trust.
* **Fine-Tuning:** The model is fine-tuned with medical dialogue datasets for better domain adaptation.
* **Example:** Given symptoms like "abdominal swelling, jaundice," the chatbot not only identifies liver disease as a likely cause but also suggests relevant lab tests and home care advice, citing specific references.

This hybrid approach leverages the power of retrieval for factual accuracy and generation for natural dialogue, making the chatbot both informative and conversational.

**Summary**

By integrating regression for statistical symptom analysis, LSTM for temporal pattern recognition, and RAG for knowledge-grounded response generation, the chatbot achieves a high level of diagnostic accuracy and user engagement. This modular, multi-model design allows the system to scale and adapt to complex medical data, ensuring users receive reliable and personalized health advice.

**Suggested Images for This Section**

* Flowchart of model pipeline: Symptom input → Regression → LSTM → RAG retrieval → Response generation.
* Diagram of LSTM cell structure showing gates and memory flow.
* Illustration of RAG architecture showing retriever fetching documents and generator producing answers.
* Sample output snippets comparing regression-only vs. RAG-enhanced chatbot responses.

**VI. IMAGE-BASED DIAGNOSIS MODULE**

**6.1 Role of Image Processing**

In the medical diagnosis context, visual data such as X-rays, MRIs, CT scans, or dermatological images hold critical diagnostic information that complements symptom-based analysis. The image-based diagnosis module enhances the chatbot’s capability by:

* Allowing users to upload medical images related to their symptoms (e.g., skin lesions, swelling, scans).
* Enabling the system to analyze these images for visual markers of diseases.
* Providing an additional evidence layer to support or refine the chatbot’s textual diagnosis.

Image processing techniques help detect patterns, anomalies, or pathological features invisible through text data alone. This multimodal approach improves diagnostic accuracy and user trust by combining symptom narratives with visual clinical evidence.

**6.2 GROQ API Integration**

For effective image-based diagnosis, this project integrates the GROQ API — a powerful AI model platform specialized in multimodal data understanding, including advanced image analysis.

* **Why GROQ?** It supports both image and natural language inputs simultaneously, enabling the model to analyze medical images in the context of user queries.
* **Integration Details:**
  + Users upload images via the chatbot frontend.
  + Images are base64-encoded and sent to the backend API.
  + The FastAPI backend forwards the encoded image alongside user symptom queries to the GROQ API.
  + The GROQ API processes the image using large multimodal models (e.g., meta-llama/llama-4-maverick-17b-128e-instruct), generating diagnostic insights related to the visual data.
* **Benefits:**
  + Real-time image interpretation without needing heavy local computation.
  + Combined text+image understanding leads to context-aware diagnostics.
  + Supports a wide range of medical image types, adaptable to various specialties.

This integration ensures that the chatbot doesn’t just rely on user-described symptoms but also harnesses the power of visual clinical data.

**6.3 Prompt Engineering with Images**

A critical technical challenge in image-based diagnosis lies in designing effective prompts for the GROQ API to maximize relevant outputs.

* **Prompt Structure:**
  + The prompt includes the user’s textual query describing symptoms.
  + The image is embedded in the prompt as a base64-encoded data URI (data:image/jpeg;base64,...).
  + The prompt asks the model to interpret both inputs jointly, e.g., “Given the symptoms and this medical image, what is the likely diagnosis?”
* **Engineering Goals:**
  + Ensure clarity so the model understands it’s a medical diagnostic task.
  + Balance between brevity and enough context for accurate reasoning.
  + Avoid ambiguous or open-ended instructions that might confuse the model.
* **Example Prompt:**

json

Copy code

[

{

"role": "user",

"content": [

{"type": "text", "text": "Patient reports abdominal swelling and jaundice."},

{"type": "image\_url", "image\_url": {"url": "data:image/jpeg;base64,<encoded\_image>"}}

]

}

]

* **Outcome:** The model combines symptom data with visual cues to generate a diagnosis or suggest further tests.

Effective prompt engineering enhances the chatbot’s multimodal reasoning and increases confidence in its medical assessments.

**6.4 Sample Input/Output Examples**

To demonstrate the module’s practical utility, consider these examples:

* **Example 1:**
  + **Input:** Image of a skin rash + symptom query: “Patient has red, itchy patches on arms and legs.”
  + **Output:** “The rash patterns and distribution suggest eczema or allergic dermatitis. Recommend dermatological consultation and possible allergy tests.”
* **Example 2:**
  + **Input:** Abdominal ultrasound image + symptom query: “Patient reports abdominal swelling and jaundice.”
  + **Output:** “Ultrasound indicates possible ascites and liver irregularities consistent with cirrhosis. Advise liver function tests and specialist referral.”
* **Example 3:**
  + **Input:** Chest X-ray image + symptom query: “Patient has persistent cough and fever.”
  + **Output:** “X-ray shows signs of pneumonia in the right lung. Immediate antibiotic treatment recommended.”

These examples illustrate how image analysis enriches symptom-based diagnosis, providing actionable insights grounded in visual evidence.

**Suggested Images for This Section**

* Flow diagram showing user image upload → base64 encoding → GROQ API request → response integration.
* Screenshot or mockup of chatbot interface with image upload feature.
* Visual examples of medical images analyzed (e.g., X-ray, skin lesion) alongside chatbot diagnostic output.

**VII. IMPLEMENTATION**

**7.1 Input Processing Flow**

The implementation of the symptom-based medical diagnosis chatbot begins with an efficient input processing flow that ensures accurate data capture and preparation for analysis.

* **User Input Collection:**  
  The frontend UI collects user inputs in two forms:
  + **Symptom Text:** The user enters symptoms or descriptions related to their health concerns.
  + **Image Upload (Optional):** Users can upload medical images such as skin lesions, scans, or X-rays to support their symptom description.
* **Data Validation:**
  + Text inputs are checked for emptiness and reasonable length.
  + Uploaded images undergo format validation (JPEG, PNG) and corruption checks using PIL in the backend.
* **Preprocessing:**
  + Text data is cleaned to remove unnecessary whitespace or special characters.
  + Images are base64 encoded for safe transmission to the AI API.
* **Request Packaging:**  
  The processed text and image (if provided) are bundled into a single JSON request structured for the AI backend, particularly the GROQ API in the case of images.

This flow ensures that only valid, sanitized inputs reach the AI processing layer, reducing errors and improving diagnosis quality.

**7.2 Integration Between Frontend, Backend, and AI**

This project’s architecture relies on seamless integration between the user-facing frontend, backend service, and AI-powered diagnosis engines.

* **Frontend:**  
  Built using React/Vue (based on your choice), the frontend presents an intuitive chat interface with:
  + Symptom input box.
  + Image upload component.
  + Real-time display of chatbot responses.
* **Backend (FastAPI):**
  + Handles API endpoints for receiving user inputs.
  + Validates and preprocesses data.
  + Manages communication with external AI APIs (OpenAI for RAG chatbot, GROQ for image diagnosis).
  + Maintains session data and user authentication.
* **AI Interaction:**
  + The backend formats inputs into prompts and sends requests to:
    - **RAG Chatbot API** for symptom-based queries.
    - **GROQ API** for multimodal image+text diagnosis.
  + Responses are parsed and returned to the frontend for display.
* **Session Management:**  
  User chat history is stored in MongoDB with vector embeddings for efficient retrieval and contextual continuity.

This tri-layer integration ensures a smooth user experience where textual and visual medical data are analyzed and responded to cohesively.

**7.3 MongoDB Vector Search with Session Chat**

A key innovation in this project is leveraging vector search within MongoDB to maintain intelligent, context-aware conversations.

* **Why Vector Search?**  
  Traditional keyword search is inadequate for medical queries that involve synonyms, varying symptom descriptions, and evolving context. Vector search embeds chat messages into high-dimensional numerical vectors, capturing semantic meaning.
* **Implementation:**
  + Each user message and chatbot response is converted to vector embeddings using pretrained models.
  + These embeddings are stored in MongoDB’s vector index, linked to the user session.
* **Session Continuity:**
  + When a user sends a new message, the system queries the vector database for semantically similar past messages.
  + Retrieved messages provide context to the RAG chatbot, improving relevance and coherence in replies.
* **Benefits:**
  + Maintains personalized conversation flow.
  + Handles ambiguous or follow-up questions by referencing previous chats.
  + Enables faster and more accurate retrieval of medical knowledge snippets.

This vector-based session chat system significantly enhances user engagement and the chatbot’s diagnostic reliability.

**Suggested Images for This Section**

* Flowchart showing end-to-end input processing from frontend input → backend validation → AI API call → frontend response display.
* Diagram illustrating the system architecture with frontend, backend, AI APIs, and MongoDB vector search.
* Visualization of vector embeddings stored in MongoDB and how similarity search works within a session.

**VIII. SYSTEM DESIGN**

**8.1 System Architecture Diagram**

The system architecture presents the overall structural layout of the Symptom-Based Medical Diagnosis Chatbot, highlighting the interaction between different components.

* **Frontend Interface:**  
  Provides the user with a friendly chat UI for entering symptoms and uploading images. It communicates with the backend through REST API calls.
* **Backend Server (FastAPI):**  
  Acts as the core processor that manages user requests, validates inputs, and handles communication with AI services.
* **AI Services:**
  + **RAG Chatbot Module:** Uses Retrieval-Augmented Generation to analyze symptom text and provide contextual diagnosis.
  + **Image-Based Diagnosis Module:** Integrates with the GROQ API for analyzing medical images to assist in diagnosis.
* **Database (MongoDB):**  
  Stores user sessions, chat histories, and vector embeddings to maintain conversation context and enable efficient information retrieval.
* **Vector Search Engine:**  
  Enables semantic search on chat history to improve response relevance by feeding context into the RAG chatbot.

This layered architecture ensures modularity, scalability, and seamless data flow between components.

**8.2 Use Case Diagram**

The use case diagram maps out key interactions between the user and the system, emphasizing primary functionalities:

* **User:**
  + Inputs symptoms via chat.
  + Uploads medical images for diagnosis.
  + Receives medical suggestions and recommendations.
  + Reviews chat history and previous diagnoses.
* **System:**
  + Validates user inputs.
  + Processes text queries via RAG chatbot.
  + Processes images via GROQ API.
  + Stores session data and retrieves relevant past chats.
  + Provides diagnostic responses and suggested actions.

The diagram visually clarifies how users engage with the system and how the backend supports those interactions.

**8.3 Class Diagram**

The class diagram models the main data structures and classes involved in the system’s backend.

* **User:**  
  Contains attributes like userID, name, authentication tokens, and chat session data.
* **ChatSession:**  
  Maintains sessionID, timestamps, linked userID, and stores messages and vector embeddings.
* **Message:**  
  Stores messageID, message content (text/image), message type, timestamp, and associated session.
* **DiagnosisResult:**  
  Contains diagnosis text, confidence scores, associated symptoms, and image analysis results.
* **APIHandler:**  
  Manages communication with RAG and GROQ AI APIs, including request formatting and response parsing.

This diagram helps in understanding how data flows and is structured programmatically within the system.

**8.4 Activity Diagram**

The activity diagram demonstrates the dynamic workflow of user interactions and system processes:

1. User opens the chatbot and inputs symptoms or uploads an image.
2. System validates input and preprocesses it.
3. Depending on input type, sends requests to the respective AI service (text to RAG, image to GROQ).
4. AI services process input and return diagnostic results.
5. Backend stores session data and embeddings in MongoDB.
6. Response is delivered to the user via frontend.
7. User can continue conversation or end session.

The diagram depicts sequential steps and decision points to visualize system behavior during diagnosis.

**8.5 Flowchart**

The flowchart offers a simplified step-by-step representation of the entire medical diagnosis process, from user input to final output:

* Start → User input (symptom/image) → Input validation
* Decision: Is input valid?
  + No → Error message → End
  + Yes → Preprocessing → API request (RAG or GROQ)
* Receive AI response → Store chat and embeddings → Display response to user
* Continue or end chat?

This provides a clear, easy-to-follow guide on how the system operates internally.

**Suggested Images for This Section**

* **System Architecture Diagram:** A block diagram showing frontend, backend, AI APIs, database, and vector search engine.
* **Use Case Diagram:** Visual actors (User) and use cases (input symptoms, upload image, receive diagnosis).
* **Class Diagram:** UML-style diagram with classes and relationships (User, ChatSession, Message, DiagnosisResult, APIHandler).
* **Activity Diagram:** Flow of user interaction and system response, with decision nodes.
* **Flowchart:** Stepwise process flow from input to diagnosis delivery.

**IX. MODULE DESCRIPTION**

**9.1 Data Collection**

Data collection is the foundational step where raw data is gathered to train and test the AI models. For this project, two types of data are collected:

* **Symptom Data:**  
  Clinical datasets containing patient symptoms and corresponding diagnoses. These include textual symptom descriptions like “abdominal swelling,” “jaundice,” etc. The data is sourced from medical records, open health datasets, and publicly available symptom-disease mapping databases.
* **Medical Images:**  
  Images such as X-rays, ultrasounds, or dermatological photos that support diagnosis. These images are gathered from medical repositories or generated during the testing phase for image-based diagnosis.

Proper data collection ensures diversity and accuracy in training data, which is critical for the chatbot’s reliability.

*Possible image:* Diagram showing sources of data (text symptom datasets, medical image repositories).

**9.2 Data Preprocessing**

Raw medical data is often noisy, incomplete, or inconsistent. Preprocessing prepares this data for efficient training:

* **Text Data:**
  + Tokenization: Splitting symptom descriptions into individual words or phrases.
  + Cleaning: Removing irrelevant symbols, correcting misspellings, and handling missing entries.
  + Standardization: Converting medical terms to a consistent format, e.g., “fever” and “pyrexia” standardized.
  + Stopwords removal: Filtering out non-informative words like “and,” “or,” to focus on key symptoms.
* **Image Data:**
  + Resizing: Normalizing image dimensions for model compatibility.
  + Noise reduction: Filtering artifacts or background noise.
  + Normalization: Adjusting pixel intensity values to a uniform scale.

Preprocessing ensures the AI models receive clean, meaningful input to improve learning efficiency and accuracy.

*Possible image:* Flowchart illustrating preprocessing steps for text and images.

**9.3 Feature Extraction**

Feature extraction transforms raw data into meaningful numerical representations that AI models can process:

* **From Text:**  
  Techniques like TF-IDF (Term Frequency-Inverse Document Frequency), word embeddings (e.g., Word2Vec, GloVe), or transformers are used to encode symptom descriptions into dense vector representations that capture semantic meaning.
* **From Images:**  
  Extracting features like shapes, edges, textures, or learned features from convolutional neural networks (CNNs) to represent image content relevant for diagnosis.

Effective feature extraction helps the models understand complex patterns between symptoms/images and diseases.

*Possible image:* Diagram showing text vectorization and image feature maps.

**9.4 LSTM Training**

Long Short-Term Memory (LSTM) networks are used to model sequential patterns in symptom data:

* **Why LSTM?**  
  Symptoms often appear in sequences, and their temporal or contextual relationships can be important in diagnosis. LSTM’s ability to remember long-term dependencies makes it ideal.
* **Training Process:**  
  The preprocessed and vectorized symptom data is fed into the LSTM network. The model learns to predict disease labels based on symptom sequences. Hyperparameters such as epochs, batch size, learning rate, and layer sizes are optimized for best performance.
* **Validation:**  
  The model is validated using a separate dataset to check accuracy and avoid overfitting.

LSTM training helps the system predict diseases more accurately by capturing symptom progression patterns.

*Possible image:* LSTM network architecture with input sequence and output prediction.

**9.5 Prediction and Diagnosis**

This module integrates outputs from both text-based LSTM predictions and image-based analysis:

* The chatbot uses the LSTM model’s predicted probabilities to identify likely diseases based on symptoms.
* Image-based diagnosis results from the GROQ API are combined to refine the prediction, especially for visual symptoms or signs.
* The RAG chatbot then generates an easy-to-understand diagnostic message for the user, combining all inputs and medical knowledge.

The final output includes the predicted disease, confidence scores, suggested next steps (e.g., tests, consultations), and possible home remedies or advice.

*Possible image:* Workflow diagram showing integration of LSTM predictions, image analysis, and chatbot response generation.

**Summary of Images to Prepare**

* Data collection sources diagram
* Preprocessing flowchart (text + image)
* Feature extraction visualization (text vector and image features)
* LSTM network architecture diagram
* Prediction integration workflow

**X. SYSTEM TESTING**

**10.1 Testing Strategy**

Testing is a critical phase of the project to ensure that the chatbot performs reliably, accurately, and consistently across different inputs and environments. The testing strategy adopted in this project is structured to cover all key modules—symptom input processing, LSTM-based prediction, RAG-based chatbot response generation, image diagnosis module, and vector-based context retrieval.

**Testing Objectives:**

* To validate the accuracy of disease prediction based on symptoms.
* To check the correct integration between the frontend, backend, ML models, and the database.
* To ensure image-based diagnosis works seamlessly using the GROQ API.
* To maintain consistent chat session history and contextual relevance using MongoDB vector search.
* To identify any bugs, logical errors, or performance bottlenecks in real-time chatbot usage.

**Testing Phases Followed:**

1. **Development Testing** – Conducted during model training and backend logic setup.
2. **Module-Level Testing** – Each module (e.g., LSTM, RAG, image API) is tested individually.
3. **System Integration Testing** – Ensures that all modules work together as expected.
4. **End-to-End Testing** – Simulates real-world scenarios from symptom entry to final response.
5. **User Acceptance Testing (UAT)** – Tested by sample users for feedback and usability improvements.

*Suggested Image:* Testing strategy pyramid or flowchart from Unit Testing → Integration → System Testing → UAT.

**10.2 Types of Testing (Unit, Integration, System)**

To ensure comprehensive testing coverage, the following testing types were applied:

**1. Unit Testing**

Unit testing focuses on validating individual functions, components, or methods in isolation.

**Examples in this project:**

* Validating symptom keyword extractor function.
* Testing accuracy of LSTM prediction module given sample inputs.
* Testing output structure of GROQ API for medical image queries.
* Ensuring response formatting and message generation from the RAG pipeline.

**Tools Used:**  
pytest, unittest (Python), manual logging for API responses.

**2. Integration Testing**

Integration testing checks if different modules and components work well together when combined.

**Integration Points:**

* Symptom input → Preprocessing → LSTM → RAG → Chatbot response.
* Frontend (React) → Backend (FastAPI) → MongoDB vector store.
* Image Upload Module → GROQ API → Response handling in chatbot.
* Session-based vector search → Chatbot → Rendered history on frontend.

**Purpose:**

* Ensure data flow is uninterrupted.
* Validate error handling and fallbacks between modules.
* Confirm smooth transition from one module to another without loss of data or context.

*Example Scenario:*  
User submits “abdominal pain, yellow eyes” → processed in backend → prediction by LSTM → chatbot responds accurately.

**3. System Testing**

This is a holistic test of the entire application to ensure all components function as a unified product.

**Covered Scenarios:**

* Normal symptom input with known diagnosis.
* Invalid/missing symptoms → chatbot handles gracefully.
* Image-based prediction in parallel with symptom-based response.
* Chat sessions remembered even on page reload.
* Real-time query handling speed and stability.
* Response accuracy for multi-symptom complex queries.

**System Testing Goals:**

* Accuracy: Validate medical logic and response relevance.
* Robustness: Ensure system doesn’t break under edge cases.
* Usability: Test chatbot interface responsiveness and UX.
* Compatibility: Cross-browser and device testing.

**Tools Used:**

* Manual Testing (real user symptom entry).
* Postman (API testing).
* Browser DevTools (frontend behavior and error checks).
* MongoDB Atlas dashboard (query and session validation).

**Summary**

Testing played a major role in validating the project’s functionality and ensuring a smooth, medically relevant, and technically accurate chatbot system. By employing various testing layers, the system was refined to deliver real-time, trustworthy symptom-based medical suggestions, integrated with LLM reasoning and contextual memory.

**Suggested Images for This Section:**

* Testing lifecycle diagram (Unit → Integration → System).
* Example test case table (input, expected output, result).
* Screenshot of chatbot response to valid/invalid input.
* Postman/API test logs (optional for appendix).

**XI. CONCLUSION AND FUTURE WORK**

**Conclusion**

This project, titled **"Symptom-Based Medical Diagnosis Chatbot Using RAG and Machine Learning,"** successfully demonstrates the fusion of advanced Natural Language Processing (NLP), Machine Learning (ML), and Retrieval-Augmented Generation (RAG) techniques to create a reliable and user-friendly AI medical assistant.

The system allows users to input symptoms in natural language and receive intelligent, medically-reasoned predictions. By using **LSTM** for learning symptom patterns and **RAG-based chatbot** architecture for contextual reasoning and medical recommendations, the solution bridges the gap between raw symptom inputs and understandable diagnoses. Additionally, the integration of **GROQ API for image analysis** expands the diagnostic capabilities to include visual symptom interpretation, such as skin rashes or swollen areas.

The project also demonstrates how **MongoDB’s vector search** can be used to create a persistent, session-aware medical chatbot that maintains context across interactions—similar to modern generative AI assistants like ChatGPT.

In terms of implementation, the backend, ML models, and database were integrated with a responsive frontend using **MERN stack technologies**, enabling smooth interaction, accurate results, and real-time feedback.

The system has been tested with both valid and invalid inputs to ensure it handles all scenarios—gracefully reporting unknown diseases for improper symptoms and giving confident, medically-sound results when given clear symptoms.

Thus, the project proves that AI-based medical diagnosis tools can be built in a scalable, modular, and intelligent manner to assist users in understanding their health concerns early on.

**Future Work**

While the current chatbot achieves high accuracy and usability, there are several directions in which the system can be enhanced in future versions:

1. **Multilingual Support:**
   * Adding support for regional languages like Tamil, Hindi, Telugu, etc., to expand accessibility for non-English speakers.
2. **Real-Time Doctor Assistance:**
   * Integration with a telemedicine platform to connect users directly to certified doctors for further consultation.
3. **Medical Record Storage:**
   * Secure user account system to store previous chats, reports, and upload medical documents (like test reports, scans) for long-term tracking.
4. **Voice Input and Output:**
   * Adding speech-to-text and text-to-speech functionality to allow hands-free interaction with the chatbot.
5. **More Deep Learning Models:**
   * Incorporating CNNs for image classification and diagnosis of X-rays, CT scans, or skin conditions.
   * Exploring transformer-based models (like BioBERT or ClinicalBERT) for even deeper medical text understanding.
6. **Integration with IoT Devices:**
   * Connecting with smartwatches or medical devices (BP monitors, glucometers) to include real-time vitals into diagnosis suggestions.
7. **Clinical Data Validation:**
   * Collaborating with medical professionals or using real clinical data to validate and fine-tune the model performance.
8. **Mobile App Version:**
   * Creating a cross-platform mobile version of the chatbot using Flutter or React Native for easier access.
9. **Emergency Alert System:**
   * If critical symptoms are detected, the chatbot can be programmed to suggest emergency services or nearest hospitals.

**Closing Statement**

This project marks a significant step toward intelligent, accessible healthcare assistance. It combines **machine learning, AI, and practical software engineering** to deliver a tool that can potentially assist millions in early symptom analysis and encourage timely medical attention. With further enhancements and real-world deployment, this chatbot could become a reliable companion in the healthcare domain.

**GLOSSARY**

| **Term** | **Description** |
| --- | --- |
| **RAG** | **Retrieval-Augmented Generation** – Combines document retrieval (from vector DBs) with language models to generate more accurate and contextually rich responses. Used in this chatbot to generate diagnoses with medical reasoning. |
| **LSTM** | **Long Short-Term Memory** – A type of recurrent neural network (RNN) that can learn long-term dependencies. Used here to learn symptom patterns from medical datasets. |
| **Vector DB** | **Vector Database** – Stores data (such as symptom embeddings or text embeddings) as high-dimensional vectors. Used for semantic search in this project with MongoDB’s vector search. |
| **GROQ API** | An advanced API service from GroqCloud that runs powerful large models like **Meta Llama 4** for fast image + text processing. |
| **Prompt Engineering** | The technique of designing inputs (prompts) to guide AI models to generate accurate, relevant outputs. |
| **Embeddings** | Dense vector representations of data (text/images). Used to find semantic similarity between user input and stored documents. |
| **MongoDB Atlas** | Cloud database platform that supports traditional NoSQL operations and advanced features like vector search. |
| **Session Storage** | Technique for maintaining conversation history across different user interactions. Used in this chatbot to remember context. |
| **Frontend** | The part of the application the user sees and interacts with. Built using React.js and styled with Tailwind CSS. |
| **Backend** | The server-side logic, models, and APIs built using FastAPI (for AI services) and Node.js (for main app logic). |

**APPENDIX A – Sample Dataset / Image Input (Page 31)**

**🧬 Symptom Dataset (Sample Rows)**

| **Symptom 1** | **Symptom 2** | **Symptom 3** | **Disease** |
| --- | --- | --- | --- |
| Fatigue | Headache | Weight Loss | Diabetes |
| Jaundice | Abdominal Pain | Dark Urine | Liver Disease |
| Chest Pain | Breathlessness | Sweating | Heart Attack |

Source: Preprocessed public health dataset (Kaggle / Custom CSV)

**🖼️ Image Input Example**

* **Input:** Swollen knee image uploaded by user
* **Prompt:** “Analyze the image and suggest possible joint issues”
* **Model Response (via GROQ API):** “The image shows signs of inflammation around the knee joint, indicating possible arthritis or fluid accumulation (effusion).”

**APPENDIX B – Screenshots of Chatbot UI & Image Module (Page 32)**

⚠️ *Add your own screenshots here* (placeholders below):

1. **Chatbot UI Sample:** Showing symptom-based conversation
2. **Invalid Symptom Handling:** “Unknown Disease - No valid symptoms provided.”
3. **Image Module UI:** Upload image and get result from GROQ
4. **Sidebar Session History:** Display previous chat history

**APPENDIX C – User Guide (How to use chatbot & image tool) (Page 33)**

**💬 How to Use the Symptom-Based Chatbot**

1. Go to the **chatbot page** on the web app.
2. Enter symptoms in plain English (e.g., *“fever, cough, fatigue”*).
3. Click “Analyze” – The chatbot responds with a probable disease and reasoning.
4. Optionally ask for a **diet plan, prescription, or home remedy**.

**🖼️ How to Use the Image-Based Diagnosis Tool**

1. Navigate to the **image tool tab**.
2. Upload an image of a visible symptom (skin rash, swelling, wound).
3. The tool sends it to GROQ + LLaMA for analysis.
4. Output shows a medical explanation with possible conditions.

**REFERENCES (Page 34)**

1. Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*.
2. Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *arXiv preprint arXiv:2005.11401*.
3. MongoDB Vector Search Documentation – <https://www.mongodb.com/docs/atlas/>
4. GROQ AI API Documentation – https://console.groq.com/
5. Meta LLaMA Model Card – HuggingFace Model Repo
6. FastAPI Official Docs – https://fastapi.tiangolo.com
7. Kaggle Symptom Dataset – <https://www.kaggle.com/>
8. ReactJS Official Documentation – <https://reactjs.org>
9. Tailwind CSS Docs – <https://tailwindcss.com/docs>