رواد مصر الرقمية

Digital Egypt Pioneers Initiative - DEPI

Graduation Project

Project name: **SympAI**

Group: **CAI1_AIS1_S2e**

Team Members:

Mohamed Gamal Kenawy
Mohamed Kamal Mohamed
Mohamed Gamal Mohamed
Mohamed Maged Mohamed
Mohamed Ahmed Abdel-Fattah

Supervisors:

Eng. Abdelrhman Ahmed

Year : 2024

Dedication

This project is dedicated to all the healthcare professionals who tirelessly strive to improve patient outcomes and make a difference in people's lives. Your commitment to caring for others inspires us to harness technology in ways that enhance accessibility and engagement in healthcare.

We also dedicate this work to the patients who seek guidance and support in navigating their health journeys. Your resilience motivates us to create solutions that prioritize your needs and enhance your access to care.

Finally, we dedicate this work to our families and friends for their unwavering support and encouragement throughout this journey. Your belief in our vision has fueled our determination to innovate and create impactful solutions.

Abstract

SympAI is an innovative healthcare chatbot designed to assist users in identifying symptoms and providing preliminary medical advice through conversational interactions. By leveraging advanced natural language processing (NLP) techniques and open-source large language models (LLMs), SympAI enhances patient engagement and accessibility in healthcare.

To optimize for budget and time constraints, we utilized pre-trained models, specifically BioMistral 7B and Meditron 7B, chosen for their effectiveness in medical reasoning. Model quantization techniques were implemented to enable efficient local deployment on limited hardware.

The implementation phase focused on developing in-context learning capabilities for multi-turn interactions, facilitated by the LangChain framework to manage conversation memory. Additionally, various AWS services, including AWS Amplify for the web application, were employed to support deployment and scalability.

SympAI aims to improve initial diagnosis for individuals seeking medical advice while supporting healthcare providers in delivering enhanced patient care, showcasing the potential of AI-driven solutions in the healthcare sector.

Acknowledgments

We would like to express our sincere gratitude to the **Digital Egypt Pioneers Initiative (DEPI)** for providing us with the invaluable opportunity to participate in the AWS Machine Learning Engineer course. This initiative has significantly enhanced our knowledge and skills in the field of machine learning.
We extend our heartfelt thanks to our instructor, **Abdelrhman Ahmed**, for his expert guidance and support throughout the course. His insights and teaching have greatly enriched our learning experience.

we appreciate everyone involved in creating and facilitating this opportunity, ensuring that we could benefit from such a transformative educational experience.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Definition

SympAI aims to address the challenge of pre-consultation in healthcare by providing an AI-driven chatbot capable of assessing patient symptoms and guiding them towards appropriate care. The system focuses on natural language processing (NLP) techniques to understand patient inputs and provide recommendations.

1.2 Motivation

The motivation behind SympAI stems from the growing need for accessible, immediate healthcare guidance, especially in situations where medical professionals may not be readily available. This project aims to streamline the early stages of patient consultation and reduce the burden on healthcare systems.

Additionally, SympAI is designed to make healthcare support more easily accessible in regions or countries where medical resources are scarce, or where the cost of healthcare is prohibitively high. By providing a reliable and affordable pre-consultation solution, SympAI can empower individuals in underserved communities to get timely healthcare advice, potentially reducing the strain on limited medical infrastructure and making healthcare more equitable.

1.3 Objectives

The main objectives of SymptomSense include:

- Providing patients with a pre-consultation platform for symptom assessment.

- Assist healthcare providers by reducing non-emergency visits through accurate early symptom detection.
- Ensuring rapid response times and user-friendly interaction.
- Allowing future integration with healthcare systems such as EMR.

1.4 Features

SympAI offers a variety of features designed to provide efficient and user-friendly healthcare support. These features include:

- **Symptom Analysis**: SympAI analyzes the symptoms provided by users and offers possible next steps or recommendations based on medical knowledge.
- **Conversational Interface**: Users interact with the chatbot in a natural, conversational manner, making the process of symptom reporting intuitive and user-friendly.
- **Evidence-based Responses**: The chatbot provides suggestions based on established medical information, avoiding specific diagnoses but giving reliable, general advice.
- **Empathetic Feedback**: SympAI acknowledges user concerns with empathetic and supportive language, ensuring patients feel heard and cared for.
- **Accessible Healthcare Support**: By offering 24/7 chatbot availability, SympAI ensures that healthcare support is accessible, particularly in regions with limited medical services or high consultation fees.
- **Emergency Protocols**: In extreme cases, SympAI is capable of suggesting contacting emergency services or directing users to seek immediate medical assistance.
- **Symptom Analysis and Recommendations:** The system will analyze user symptoms based on text inputs and offer potential actions (e.g., self-care tips, suggesting hospital visits).
- **Natural Language Processing (NLP):** SympAI will use NLP models to understand user inputs and generate responses based on a predefined knowledge base.

- **API for Integration:** Providing a clear API for integration into hospital websites and healthcare platforms.

1.4.1 Out-of-Scope

The following aspects and functionalities are not included in the initial phase of the SympAI project:

- **Medical Diagnoses:** SympAI will not provide formal medical diagnoses. It will only offer potential next steps based on symptoms.
- **Integration with EMR Systems:** While this is planned for future releases, EMR integration is not part of the initial release.
- **Multilingual Support:** Initial versions will focus on English, with other languages considered for future updates.
- **Advanced AI-Driven Diagnostics:** SympAI won't include AI models designed to replace professional diagnostic tools or perform deep medical analysis.

Chapter 2

Analysis

2.1 Feasibility Study

In assessing the technical feasibility of the project, it became clear that a wealth of open-source models and frameworks are available for the development of our application. These resources greatly facilitate the implementation of advanced functionalities, particularly for training and fine-tuning large language models (LLMs).

However, the primary challenge lies in the hardware requirements necessary to train, fine-tune, and effectively run these LLMs. Given the substantial computational power and data storage needed, reliance on local hardware was deemed impractical. This situation naturally leads us to consider cloud solutions.

By opting for cloud services, particularly **Amazon Web Services (AWS)**, we can address both the computational and storage needs of the project. AWS offers a range of services that align with our requirements, yet the associated costs can raise concerns.

To mitigate these concerns, we plan to leverage the **AWS Free Tier** during the initial development phase. This approach allows us to utilize various AWS services without incurring immediate costs, providing an effective means to validate our system's architecture and performance before scaling up.

In summary, while the technical framework and models are accessible and viable, the project's success hinges on effectively managing hardware requirements and costs through cloud solutions, specifically by utilizing the AWS Free Tier for our development needs.

AWS Service	**Description**	**Free Tier Coverage**	**Pricing (if exceeded free tier)**
Amazon EC2	Resizable compute capacity in the cloud.	750 hours/month of t2.micro or t3.micro	$0.0116/hour (Linux), $0.013/hour (Windows)
Internet Gateway	Allows communication between instances in your VPC and the internet.	N/A	N/A
AWS Amplify	Build and deploy web applications easily.	15 GB data transfer out, 1,000 build minutes, 5 GB storage, 500,000 requests (SSR), 100 GB-hours (SSR)	$0.023/GB stored, $0.01/build minute, $0.15/1 million requests (after free tier)
Amazon DynamoDB	Fully managed NoSQL database service.	25 GB of storage, 25 write capacity units, and 25 read capacity units	$1.25 per WCU, $0.25 per RCU (after free tier)
Amazon S3	Secure, durable, and scalable object storage.	5 GB of Standard Storage, 20,000 GET Requests, 2,000 PUT Requests	$0.023/GB (first 50 TB)
Elastic Load Balancer	Automatic distribution of incoming application traffic across multiple Amazon EC2 instances.	750 hours/month shared between Classic and Application load balancers, 15 GB of data processing for Classic, 15 LCUs for Application	$0.008 per LCU-hour (Application Load Balancer), $0.008 per GB processed (Classic)
Amazon Elastic Container Registry	Store and retrieve Docker images.	50 GB of storage for public repositories, 500 MB for private repositories, 5 TB of authenticated data transfer out of public repositories, 500 GB of anonymous data transfer out of public repositories	$0.10/GB (beyond free tier for private repositories)

Table 2.1: AWS Services Used in the Project

2.2 User Requirements

In this section, we identify the needs and expectations of the users of SympAI. Understanding user requirements is crucial for designing a system that effectively addresses their concerns and provides valuable assistance.

2.2.1 Use Cases and User Stories

- **Use Case 1: Symptom Submission**
  - Users can input symptoms using a free-text format or predefined dropdown options.
  - The system will validate inputs for clarity and completeness.
- **Use Case 2: Recommendations Provided**
  - The system analyzes submitted symptoms and generates recommendations.
  - Recommendations may include self-care tips, referrals to healthcare providers, or urgent care suggestions.
- **Use Case 3: Illness Awareness**
  - The system correlates symptoms with potential illnesses or conditions and provides informational content.
  - Users can access detailed descriptions of potential illnesses.
- **Use Case 4: User Login**
  - The system provides a secure login interface using email and password.
  - Users can reset their password through a secure recovery process.
- **Use Case 5: User Registration**
  - New users can register by providing necessary information, including name, email, and password.
  - The system sends a verification email to confirm the account.
- **Use Case 6: Access Chat Sessions**
  - Users can view a history of their previous chat sessions.

  - The system allows users to revisit and review past interactions for continuity.

- **User Story 1: As a user, I want to describe my symptoms so that I can receive appropriate guidance.**
- **User Story 2: As a user, I want to understand the next steps I should take based on my symptoms.**
- **User Story 3: As a user, I want to log in to my account to access my past interactions.**

2.3 System Requirements

This section outlines the necessary requirements for the SympAI system.

2.3.1 Functional Requirements

Symptom Analysis

Req.1. The system shall analyze user symptoms and provide personalized recommendations.

Req.2. The system shall utilize Natural Language Processing (NLP) techniques to enhance understanding of user inputs.

User Management

Req.3. The system shall allow users to register for an account with validation on input fields.

Req.4. The system shall allow users to log in using secure credentials.

Req.5. The system shall provide password recovery options via email.

Req.6. The system shall allow users to update their profile information (e.g., email, password).

Chat Session Management

Req.7. The system shall store user chat sessions securely for future reference.

Req.8. The system shall allow users to delete their chat history if desired.

Req.9. The system shall provide a user-friendly interface for accessing past interactions.

API for Integration

Req.10. The system shall provide a clear API for integration into hospital websites and healthcare platforms.

Req.11. The API shall support authentication for secure access.

2.3.2 Non-Functional Requirements

Security

Req.1. The system shall implement encryption for sensitive user data, both in transit and at rest.

Scalability

Req.2. The system shall be scalable to accommodate increasing user loads.

Req.3. The architecture shall support horizontal scaling as user demand grows.

Usability

Req.4. The system shall provide a user-friendly interface with intuitive navigation.

Req.5. The system shall be accessible on various devices (desktop, mobile, tablet).

Chapter 3

Design

3.1 Overall Context Diagram

The diagram illustrates a distributed system architecture with five main components. At the center is a cloud-shaped element labeled ”Network,” representing the core communication layer of the system. This network component is connected to four other key elements:

- **Core Backend Layer:** This component is responsible for managing chat history, prompt preprocessing, providing an API for the UI layer, and user management.
- **Web Server/UI Layer:**this layer handles user interface interactions and serves as the entry point for external requests.
- **DynamoDB:** This element indicates the NoSQL database service used for storing user information and chat history.
- **LLM Layer:**This layer represents a Large Language Model, likely used for natural language processing tasks.

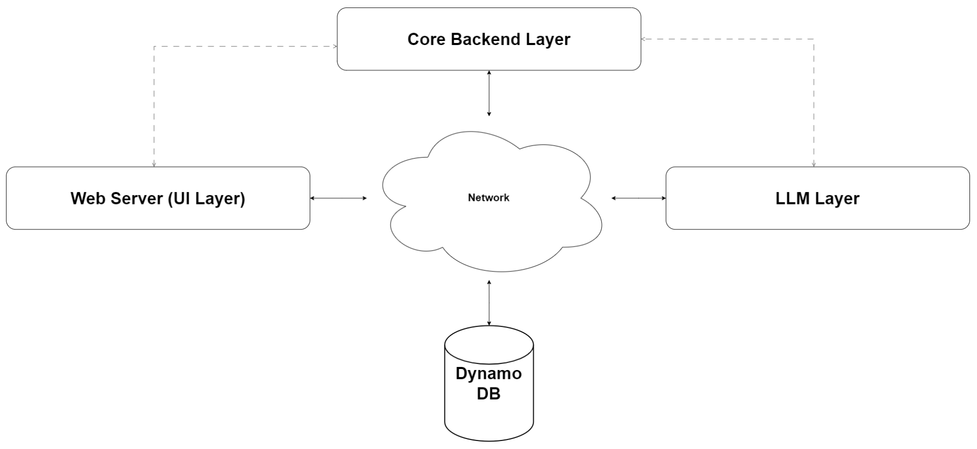


Figure 3.1: SympAI Context-Diagram Level-0

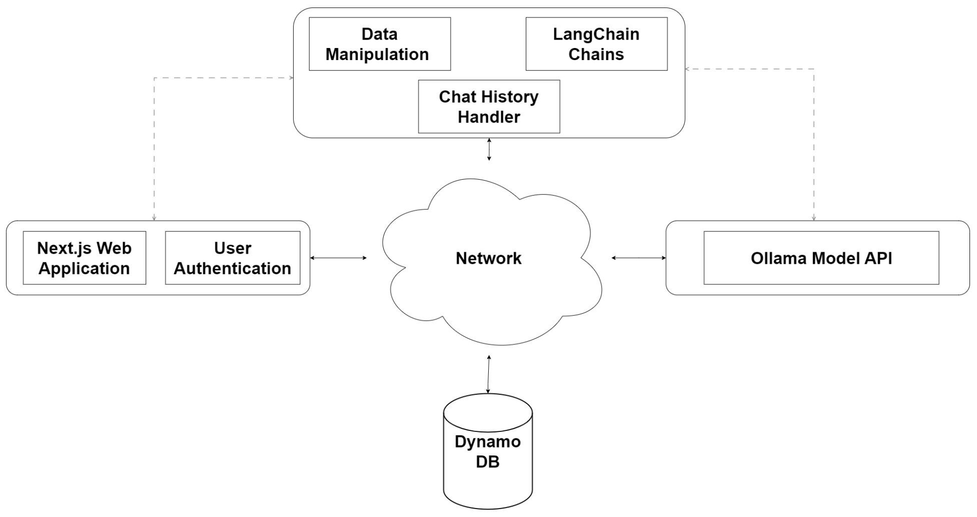
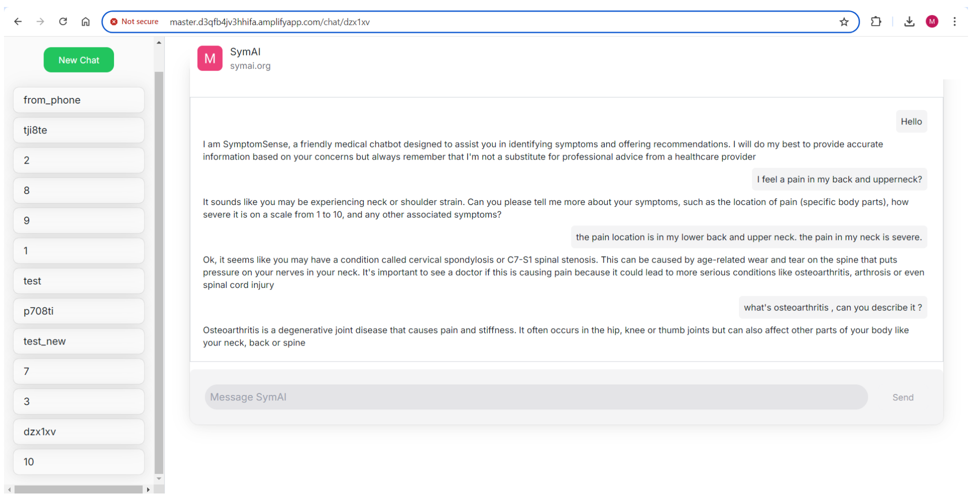


Figure 3.2: SympAI Context-Diagram Level-1

3.2 User Interface Design

This section shows displays the design for SympAI. The layout features a side-bar on the left with navigation options between chat sessions and the main chat window on the right.

3.2.1 Prototype



Hello

I am SymptomSense, a friendly medical chatbot designed to assist you in identifying symptoms and offering recommendations. I will do my best to provide accurate information based on your concerns but always remember that I'm not a substitute for professional advice from a healthcare provider

I feel a pain in my back and upperneck?

It sounds like you may be experiencing neck or shoulder strain. Can you please tell me more about your symptoms, such as the location of pain (specific body parts), how severe it is on a scale from 1 to 10, and any other associated symptoms?

the pain location is in my lower back and upper neck. the pain in my neck is severe.

Ok, it seems like you may have a condition called cervical spondylosis or C7-S1 spinal stenosis. This can be caused by age-related wear and tear on the spine that puts pressure on your nerves in your neck. It's important to see a doctor if this is causing pain because it could lead to more serious conditions like osteoarthritis, arthrosis or even spinal cord injury

what's osteoarthritis , can you describe it ?

Osteoarthritis is a degenerative joint disease that causes pain and stiffness. It often occurs in the hip, knee or thumb joints but can also affect other parts of your body like your neck, back or spine

Figure 3.3: Chat Page

3.3 AWS Solution

The flowchart illustrates the cloud solution for our project, showcasing a robust application architecture with multiple components and their interactions. Below is a detailed breakdown of the connections and flow:

1. **Initiation**: The process commences at the ”Olmala” prompt, which serves as the starting point for user interactions.

2. **Web Server**: Incoming requests are directed to the web server, which acts as the primary entry point for handling user traffic.

3. **API Gateway**: The web server forwards the traffic to the Amazon API Gateway, which functions as a rate limiter and request processor, ensuring efficient management of incoming requests.

4. **Load Balancing**: The API Gateway then routes requests to the AWS Elastic Load Balancer. This component plays a crucial role in distributing incoming traffic across multiple instances of the application, thereby enhancing performance and reliability.

5. **Backend Processing**: The Elastic Load Balancer directs the requests to a FastAPI Docker container running the backend service. This container is responsible for executing the core business logic of the application.

6. **Response Handling**: Upon processing the request, the backend service sends the response back through the API Gateway to the web server.

7. **Completion of Request Cycle**: Finally, the web server forwards the response to the user, completing the request cycle.

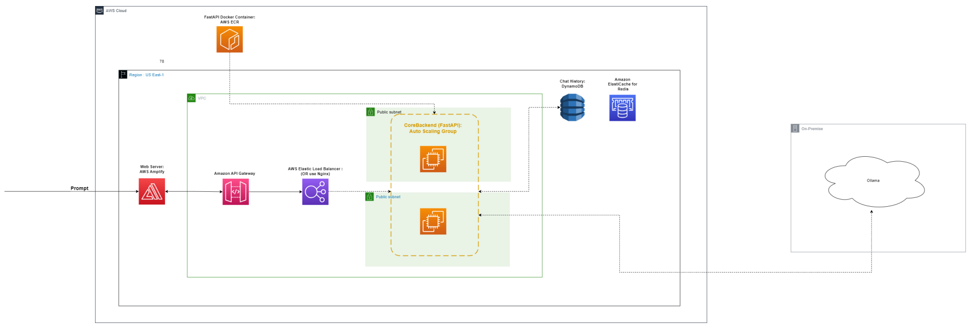

AWS Cloud
FastAPI Docker Container: AWS ECR
78
Region : US East-1
VPC
Public subnet
CoreBackend (FastAPI): Auto Scaling Group
Chat History: DynamoDB
Amazon ElastiCache for Redis
On-Premise
Ollama
Web Server: AWS Amplify
Prompt
Amazon API Gateway
AWS Elastic Load Balancer : (OR use Nginx)
Public subnet

Figure 3.4: AWS Solution

This architecture exemplifies a microservices approach, allowing each component to scale independently according to demand. The integration of AWS services, such as API Gateway and Elastic Load Balancer, provides automatic scaling and enhances resilience to accommodate varying traffic volumes.

Chapter 4

Implementation

4.1 Model Implementation and Optimization

In this section, we detail the technical steps involved in selecting, customizing, and deploying the large language model (LLM) used in our project. Given the constraints on budget, hardware, and time, we opted for open-source models, leveraging existing pre-trained LLMs. The goal was to optimize the model for our specific use case while keeping hardware requirements minimal. We also cover the implementation of in-context learning and multi-turn conversations, alongside the processes of model selection, quantization, local deployment using Ollama, and managing conversation memory with LangChain.

4.1.1 Model Selection and Fine-Tuning Challenges

Fine-tuning an LLM would have placed considerable pressure on our limited budget due to the required hardware resources. Training a model from scratch would have been even more time-consuming and resource-intensive. To avoid these issues, the first step in our model selection process was to leverage existing LLMs, specifically those pre-trained or fine-tuned on data relevant to our problem domain.

We sought a model that was open-source and had minimal hardware costs for inference, ideally one that could run on commercial-grade PCs. As such, we conducted a search on the Hugging Face Hub, which houses a wide range of open-source models. After evaluating several options, two models stood out:

- **BioMistral 7B**: A suite of Mistral-based open-source models, further pre-trained for the medical domain using data from PubMed Central. This model is fine-tuned on Mistral and optimized for medical applications.

- **Meditron 7B**: A medical LLM adapted from LLaMA-2-7B through continued pretraining on a curated medical corpus. Meditron-7B outperforms

LLaMA-2-7B on multiple medical reasoning tasks and is well-suited for our use case.

Both models were promising and small enough (7B parameters) to run on commercial processors. However, memory constraints still posed a challenge. To mitigate this, we employed model quantization.

4.1.2 Model Quantization

Model quantization is a technique that reduces the size of a model by compressing its weights without significantly sacrificing performance. Several quantization techniques were considered, including QLoRA, AWQ, and GGML/GGUF. We chose to use **GGUF** quantization with the **Q4_K_M** method, which offered the best balance between performance and memory efficiency.

Q4_K_M: is a k-quant method that uses **GGML_TYPE_Q6_K** for half of the attention.wv and feed_forward.w2 tensors, else **GGML_TYPE_Q4_K**.

- **GGML_TYPE_Q4_K**: A 4-bit quantization method that uses super-blocks, allowing for compressed weights while maintaining reasonable accuracy.
- **GGML_TYPE_Q6_K**: A 6-bit quantization method, used for a portion of the model's layers to enhance precision in key computations.

This quantization reduced the model's memory footprint, making it possible to run it on our local hardware, which consisted of:

- **CPU**: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
- **RAM**: 16 GB
- **GPU**: NVIDIA RTX 3060 with 6 GB VRAM

4.1.3 Running the Model Locally with Ollama

We used **Ollama** to run the quantized model locally. Ollama is a platform that simplifies running large language models on local hardware, offering efficient model management and inference without the need for cloud infrastructure.

Introduction to Ollama:

Ollama provides a streamlined environment for deploying LLMs on local machines, reducing the cost and complexity of cloud-based solutions. It supports various quantization formats, including GGML, making it ideal for projects with limited hardware resources. The platform also allows users to easily adjust model parameters and experiment with different configurations, making it highly adaptable to different use cases.

4.1.4 In-Context Learning and Prompt Engineering

While we did not train or fine-tune our models, we were able to customize them using in-context learning and prompt engineering techniques. We experimented with different prompt structures, including persona-based, zero-shot, and few-shot prompts, to optimize the model's output.

Below is an example of the prompt file used with the BioMistral-7B model in **Ollama** after updates:

```

FROM "./BioMistral-7B.Q4_K_M.gguf"
# set the temperature to 1 [higher is more creative, lower is more coherent]
PARAMETER temperature 0.3
PARAMETER top_p 0.9
PARAMETER top_k 40
PARAMETER repeat_penalty 1.3

# set the system message
SYSTEM """
You are SymptomSense, a friendly medical chatbot designed to assist patients in identifying symptoms and offering recommendations. Your primary goal is to provide accurate and empathetic responses to users' inquiries. Encourage users to describe their symptoms clearly and provide relevant information while ensuring a welcoming environment. Always remind users that you are not a substitute for professional medical advice and recommend consulting a healthcare professional for any serious concerns.
"""
```

```

TEMPLATE """{{ if .System }}<|start_header_id|>
  ↪ system<|end_header_id|>

{{ .System }}<|eot_id|>{{ end }}{{ if .Prompt }}<|
  ↪ start_header_id|>user<|end_header_id|>

{{ .Prompt }}<|eot_id|>{{ end }}<|start_header_id|>
  ↪ assistant<|end_header_id|>

{{ .Response }}<|eot_id|>"""
PARAMETER stop "<|start_header_id|>"
PARAMETER stop "<|end_header_id|>"
PARAMETER stop "<|eot_id|>"
```

Listing 4.1: BioMistral ModelFile

4.1.5 Handling Multi-Turn Conversations with LangChain

After setting up single-prompt queries, we needed to handle multi-turn conversations. This involved managing dialogue history and ensuring that the context was preserved across conversation turns. We used the **LangChain** framework to facilitate this.

Introduction to LangChain:

LangChain is a framework designed to enable developers to build applications that use language models. It provides essential tools for managing conversation states, such as templates for prompts, managing context, and handling multi-turn conversations. LangChain allows the integration of language models with other tools and data storage solutions, making it suitable for applications requiring state persistence, such as chatbots.

To manage chat history and preserve context, we employed the following strategies:

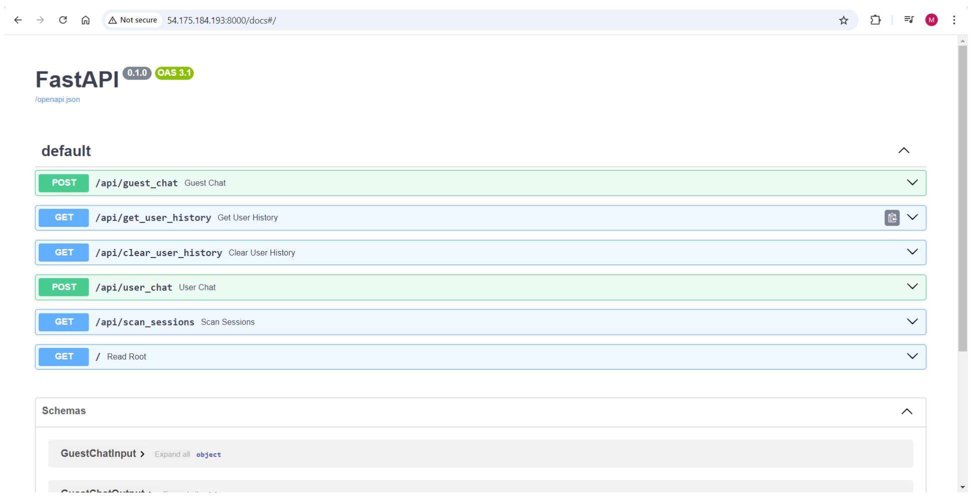
- **Chat Templates**: These templates identify the role of each participant (Human or AI) and structure conversation turns.

- **Context Management**: We saved the conversation history, keeping in mind the model's context window size.

- **Session Persistence**: Chat sessions were stored in DynamoDB for later access and continuity, leveraging AWS's free-tier offering for testing purposes.

4.2 Core API Implementation with FastAPI

The backend core API for SympAI is implemented using the **FastAPI** framework. FastAPI is a modern, high-performance web framework for building APIs, and it's chosen for its asynchronous support and automatic data validation.


Not secure 54.175.184.193:8000/docs#/

FastAPI 0.1.0 OAS 3.1
/openapi.json

default

POST /api/guest_chat Guest Chat
GET /api/get_user_history Get User History
GET /api/clear_user_history Clear User History
POST /api/user_chat User Chat
GET /api/scan_sessions Scan Sessions
GET / Read Root

Schemas

GuestChatInput › Expand all object

Figure 4.1: SympAI-API Visual Documentation

4.2.1 Key Features of the FastAPI Core

- **Asynchronous API Calls**: FastAPI supports asynchronous programming, which allows the system to handle many requests efficiently and respond faster, making it ideal for real-time applications like SympAI.
- **Automatic Documentation**: FastAPI automatically generates interactive API documentation with OpenAPI and Swagger, simplifying the development and testing process.
- **Data Validation and Serialization**: FastAPI ensures input validation, leveraging Python type hints, which makes it easy to work with incoming user data.

The core API provides endpoints for handling user authentication, retrieving chat history, managing user sessions, and sending queries to the NLP model via LangChain. FastAPI's structure allows smooth integration between the user-facing frontend and the backend NLP services.

4.2.2 Endpoints Overview

/api/guest_chat

Method: POST
Summary: Handles guest user chat prompts.
Request Body:

- `prompt` (string, required): The message from the guest, limited to 2048 characters.
- `max_tokens` (integer, optional): Maximum number of tokens to be generated (default: 4096).
- `temperature` (number, optional): Sampling temperature (example: 0.4).
- `top_p` (number, optional): Nucleus sampling parameter (example: 0.5).
- `n` (integer, optional): Number of responses to generate (example: 1).
- `presence_penalty` (number, optional): Penalty for new topic introduction (example: 0.5).
- `frequency_penalty` (number, optional): Penalty for topic repetition (example: 0.5).

Response:

- `200 OK`: Returns a response object with the generated text from the chatbot.
  Example:

```
{
  "response": "Hi. I'm SymptomSense, a medical chatbot..."
}
```

- `422 Unprocessable Entity`: Validation error.

/api/get_user_history

Method: GET
Summary: Retrieves chat history for a specific user session.
Parameters:

- `session_id` (string, required): The unique session identifier for the user.

Response:

- `200 OK`: Returns user history.
- `422 Unprocessable Entity`: Validation error.

/api/clear_user_history

Method: GET
Summary: Clears the chat history for a specific user session.
Parameters:

- `session_id` (string, required): The unique session identifier for the user.

Response:

- `200 OK`: Confirmation of cleared history.
- `422 Unprocessable Entity`: Validation error.

/api/user_chat

Method: POST
Summary: Handles registered user chat prompts.
Request Body:

- `prompt` (string, required): The user input message, limited to 2048 characters.
- `table_name` (string, required): The table for storing session information.
- `session_id` (string, required): The unique session identifier.

Response:

- `200 OK`: Returns a chatbot response.
  Example:

  ```
  {
    "response": "Hi. I'm SymptomSense, a medical chatbot..."
  }
  ```

- `422 Unprocessable Entity`: Validation error.

/api/scan_sessions

Method: GET
Summary: Retrieves a list of all active user sessions.
Response:

- `200 OK`: An array of active session IDs.
  Example:

```
[
  "session1",
  "session2"
]
```

/ (Root Endpoint)

Method: GET
Summary: Root endpoint to confirm API is operational.
Response:

- `200 OK`: A simple success message or empty response.

4.2.3 Validation Error Schema

Errors returned by the API in case of invalid requests follow the schema below:

```
{
  "detail": [
    {
      "loc": ["location", "error"],
      "msg": "Validation error message",
      "type": "validation_error_type"
    }
  ]
}
```

4.3 Dockerization of Services

To ensure consistency and ease of deployment, the entire SympAI backend and frontend are containerized using **Docker**. Docker allows each component of the system (e.g., FastAPI backend, web app, LangChain NLP service, DynamoDB) to run in isolated containers with all their dependencies pre-installed.

4.3.1 Key Benefits of Dockerization

- **Portability**: Docker containers ensure that SympAI runs identically in different environments (local development, staging, production) without compatibility issues.
- **Scalability**: Containers can be easily scaled horizontally, meaning we can increase the number of containers running the FastAPI backend or web app based on traffic demand.
- **Isolation**: Docker ensures that each service (e.g., FastAPI, web app, database) is isolated in its own environment, reducing the risk of conflicts and ensuring high security.
- **Continuous Integration and Deployment**: Docker images are integrated into the CI/CD pipeline, ensuring that the latest versions of the services are always deployed seamlessly.

4.3.2 Dockerfile for FastAPI Backend

The FastAPI backend is built and deployed using a multi-stage Dockerfile. This ensures that the final image is lightweight and includes only the necessary dependencies for running the FastAPI application.

```
# Stage 1: Build dependencies
FROM python:3.12.1-slim AS build

# Set the working directory in the build stage
WORKDIR /app

# Copy the requirements file first to leverage
   ↪ Docker's cache
COPY ./requirements.txt /app/requirements.txt

# Install dependencies (only for building stage)
```

```
RUN pip install --no-cache-dir --upgrade -r /app/
    requirements.txt

# Stage 2: Final image
FROM python:3.12.1-slim

# Set the working directory in the final stage
WORKDIR /app

# Copy only the necessary files from the build stage
COPY --from=build /usr/local/lib/python3.12/site-
    packages /usr/local/lib/python3.12/site-
    packages
COPY --from=build /usr/local/bin /usr/local/bin

# Copy the rest of the application code to the final
     image
COPY . .

# Set environment variables (use secrets for AWS in
    production)
ENV INIT_PATHS_DIR="/app/src"
ENV AWS_REGION="us-east-1"
ENV FRONT_URL="http://localhost:3000"
ENV OLLAMA_BASE_URL="http://localhost:11434"

# Expose the port FastAPI will run on
EXPOSE 8000
EXPOSE 11434

# Command to run the FastAPI app using Uvicorn
CMD ["uvicorn", "src.api.app:app", "--host", "
    0.0.0.0", "--port", "8000", "--ssl-keyfile", "
    cert/key.pem", "--ssl-certfile", "cert/cert.pem
    "]
```

Listing 4.2: Dockerfile for FastAPI Backend

4.3.3 Dockerfile for Web Application

The web application is built using a Next.js framework and is also containerized using a multi-stage Dockerfile. This Dockerfile ensures efficient dependency management and builds a minimal production-ready image.

```
FROM node:18-alpine AS base

# Install dependencies only when needed
FROM base AS deps
# Check https://github.com/nodejs/docker-node/tree/b4117f9333da4138b03a546ec926ef50a31506c3#nodealpine
# to understand why libc6-compat might be needed.
RUN apk add --no-cache libc6-compat
WORKDIR /app

# Install dependencies based on the preferred package manager
COPY package.json yarn.lock* package-lock.json* pnpm-lock.yaml* ./
RUN \
  if [ -f yarn.lock ]; then yarn --frozen-lockfile; \
  elif [ -f package-lock.json ]; then npm ci; \
  elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm i --frozen-lockfile; \
  else echo "Lockfile not found." && exit 1; \
  fi

# Rebuild the source code only when needed
FROM base AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .

RUN \
  if [ -f yarn.lock ]; then yarn run build; \
  elif [ -f package-lock.json ]; then npm run build; \
  elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm run build; \
  else echo "Lockfile not found." && exit 1; \
  fi

# Production image, copy all the files and run next
FROM base AS runner
WORKDIR /app

ENV NODE_ENV=production

RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

COPY --from=builder /app/public ./public

# Set the correct permission for prerender cache
RUN mkdir .next
RUN chown nextjs:nodejs .next

# Automatically leverage output traces to reduce image size
# https://nextjs.org/docs/advanced-features/output-file-tracing
COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static

USER nextjs

EXPOSE 3000

ENV PORT=3000

# server.js is created by next build from the standalone output
ENV HOSTNAME="0.0.0.0"
CMD ["node", "server.js"]
```

Listing 4.3: Dockerfile for Web Application

4.4 Web Application

The web application for **SympAI** is developed using the **Next.js** framework. Next.js, built on top of React, offers powerful features such as:

- **Server-Side Rendering (SSR)**: Allows pages to be rendered on the server at request time, improving performance and SEO.
- **Static Site Generation (SSG)**: Enables the generation of static HTML pages at build time, enhancing load times and reducing server workload.
- **API Routes**: Integrated API endpoints within the Next.js structure, simplifying the communication between the frontend and the backend.
- **Routing and Dynamic Pages**: Next.js provides a file-based routing system with support for dynamic routing, allowing easy management of user-driven content.

SympAI utilizes these features to create a fast, interactive user experience, enabling seamless interactions with the backend services and providing real-time feedback to users based on their inputs.

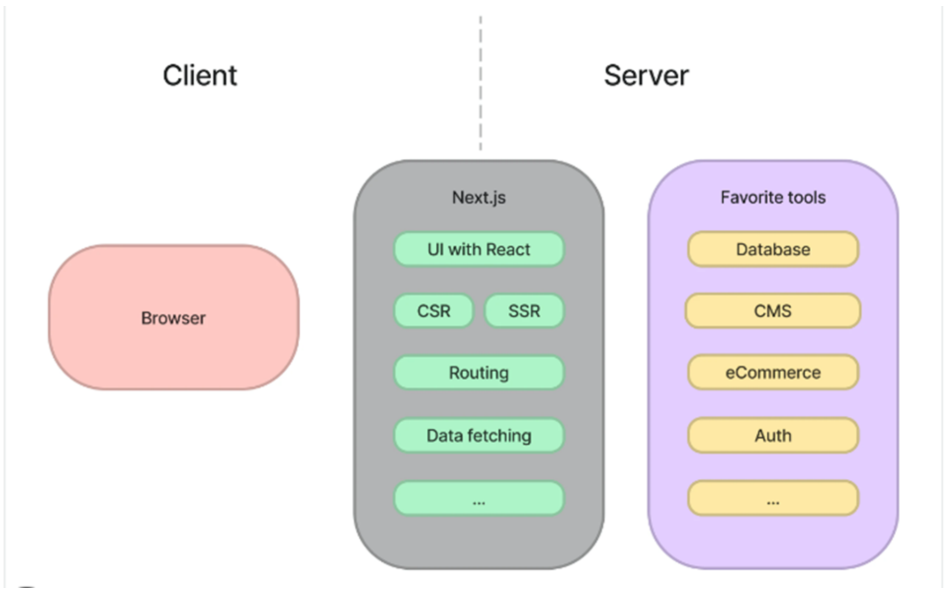


Figure 4.2: Next.js Architecture for SympAI

The **Next.js** architecture is designed with performance and scalability in mind. It provides a flexible platform for the SympAI web interface to communicate with backend services (such as the NLP models and data storage) efficiently. The combination of **SSR** and **SSG** ensures that users experience fast page loads and responsive interfaces, regardless of the complexity of their queries.

4.4.1 Key Features of the Next.js Implementation

- **Interactive UI for Symptom Input**: Users interact with an intuitive interface to input symptoms, which are then processed by the backend.
- **Dynamic Pages for User Sessions**: User sessions and chat histories are dynamically loaded using Next.js routing, ensuring personalized experiences for each user.
- **Efficient API Communication**: The web application uses Next.js API routes to efficiently send and retrieve data from the backend, ensuring real-time responses for users.

4.5 Backend Integration with NLP Models

The backend of SympAI is powered by a **Natural Language Processing (NLP)** model, which interacts with the user through a chat-based interface. This integration allows SympAI to process the user's symptom input and provide meaningful feedback or recommendations.

We leveraged the **LangChain** framework to manage the interaction between the frontend and the **LLM (Large Language Model)**. The LangChain framework simplifies the process of calling LLMs, chaining different models or tasks, and providing structured outputs.

4.5.1 LangChain Integration

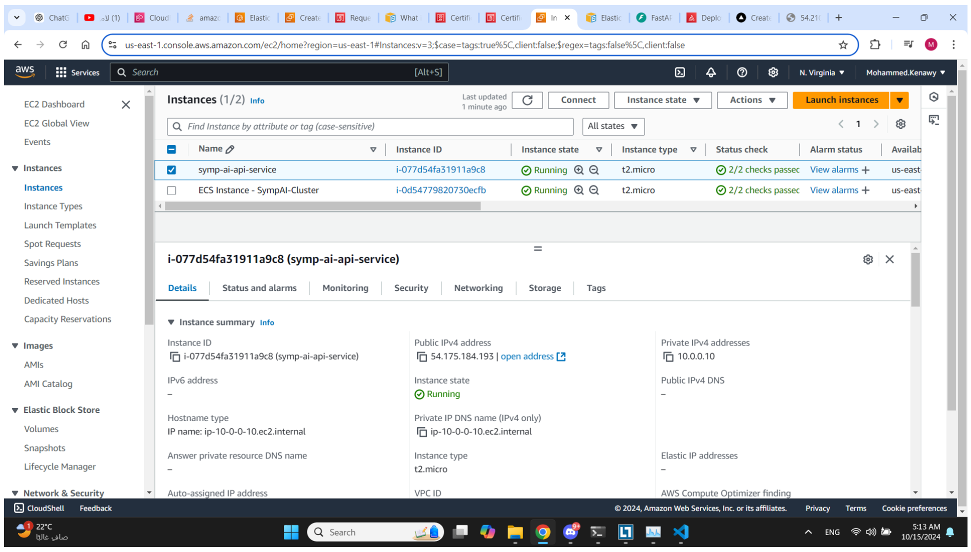
- **Prompt Engineering**: Carefully designed prompts are sent from the frontend to the backend, which passes them through LangChain to the LLM, ensuring the right context is maintained for symptom analysis.
- **Chained Responses**: LangChain allows us to process user input across multiple models or stages, improving the accuracy and relevance of the system's suggestions.
- **Response Preprocessing**: LangChain processes the LLM output before sending it back to the frontend for presentation to the user, ensuring that the responses are clear, informative, and actionable.

This integration enables SympAI to respond to user queries in a conversational style, leveraging the power of LLMs for robust natural language understanding.

4.6 AWS Infrastructure

SympAI is hosted on AWS using a variety of services, each fulfilling a specific role in the architecture:

- **AWS EC2 Instances**: The NLP models and backend services run on Amazon EC2 instances, providing scalable compute resources to handle real-time user queries.
- **DynamoDB**: DynamoDB is used to store user information, chat history, and session data. It provides a low-latency, NoSQL database solution that scales easily with user demand.
- **AWS Amplify**: AWS Amplify is used to manage the CI/CD pipeline and hosting for the SympAI web application.



us-east-1.console.aws.amazon.com/ec2/home?region=us-east-1#Instances:v=3;$case=tags:true%5C,client:false;$regex=tags:false%5C,client:false

Instances (1/2) Info
Last updated 1 minute ago

Find Instance by attribute or tag (case-sensitive)

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availab
symp-ai-api-service	i-077d54fa31911a9c8	Running	t2.micro	2/2 checks passec	View alarms +	us-east-
ECS Instance - SympAI-Cluster	i-0d54779820730ecfb	Running	t2.micro	2/2 checks passec	View alarms +	us-east-

i-077d54fa31911a9c8 (symp-ai-api-service)

Details | Status and alarms | Monitoring | Security | Networking | Storage | Tags

Instance summary Info

Instance ID	Public IPv4 address	Private IPv4 addresses
i-077d54fa31911a9c8 (symp-ai-api-service)	54.175.184.193 | open address	10.0.0.10
IPv6 address	**Instance state**	**Public IPv4 DNS**
–	Running	–
Hostname type	**Private IP DNS name (IPv4 only)**	
IP name: ip-10-0-0-10.ec2.internal	ip-10-0-0-10.ec2.internal	
Answer private resource DNS name	**Instance type**	**Elastic IP addresses**
–	t2.micro	–
Auto-assigned IP address	**VPC ID**	**AWS Compute Optimizer finding**

22°C صاف غالبًا
5:13 AM 10/15/2024

Figure 4.3: SympAI API docker image running on an ec2

4.6.1 CI/CD and Hosting with AWS Amplify

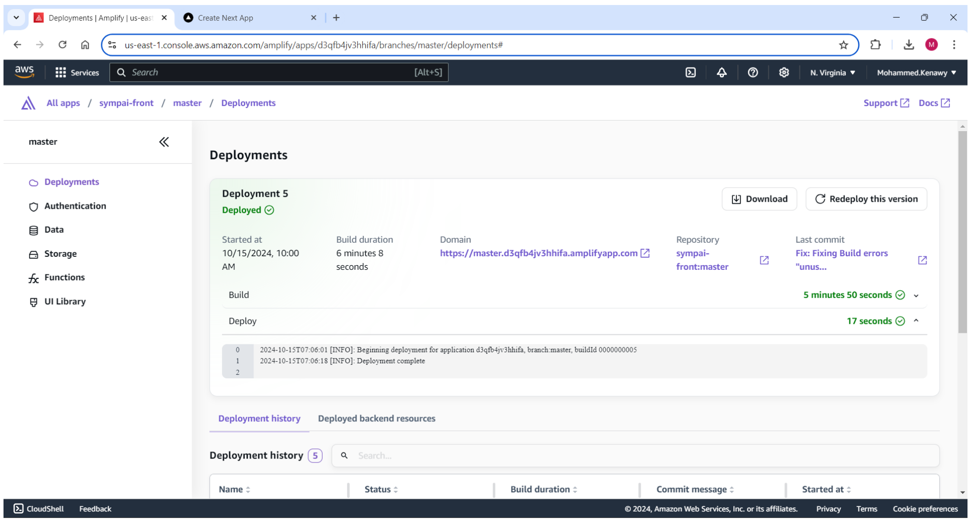
AWS Amplify provides fully managed services for the continuous integration and continuous deployment (CI/CD) of SympAI's web application. With Amplify, the deployment process is streamlined, allowing rapid updates and ensuring that the application remains available without downtime.

Key features of AWS Amplify for SympAI:

- **Continuous Deployment**: AWS Amplify automatically detects code changes from our GitHub repository, builds the application, and deploys it to a live environment. This ensures that any updates or bug fixes are reflected in real time.

- **Scalable Hosting**: The hosting service provided by AWS Amplify ensures that SympAI can scale based on traffic, handling multiple users simultaneously without performance degradation.

- **Automated Rollbacks**: In case of failed deployments or issues, Amplify provides automated rollbacks to previous stable versions, reducing downtime and minimizing the impact on users.


Deployments | Amplify | us-eas
Create Next App
us-east-1.console.aws.amazon.com/amplify/apps/d3qfb4jv3hhifa/branches/master/deployments#
Services
Search [Alt+S]
N. Virginia
Mohammed.Kenawy
All apps / sympai-front / master / Deployments
Support
Docs
master
Deployments
Authentication
Data
Storage
Functions
UI Library
Download
Redeploy this version
Deployment history
Deployed backend resources
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

Deployments

Deployment 5
Deployed

Started at: 10/15/2024, 10:00 AM
Build duration: 6 minutes 8 seconds
Domain: https://master.d3qfb4jv3hhifa.amplifyapp.com
Repository: sympai-front:master
Last commit: Fix: Fixing Build errors "unus...

Build 5 minutes 50 seconds
Deploy 17 seconds

```
2024-10-15T07:06:01 [INFO]: Beginning deployment for application d3qfb4jv3hhifa, branch:master, buildId 0000000005
2024-10-15T07:06:18 [INFO]: Deployment complete
```

Deployment history (5)

Name	Status	Build duration	Commit message	Started at

Figure 4.4: SympAI Web Application Deployed on AWS Amplify