SARVAM AI:

HOME-ASSIGNMENT DOCUMENTATION

**Acknowledgement**

I want to sincerely thank Saurabh Karn, founding member at Sarvam AI, and Nitesh Methani, machine learning engineer at the company, for giving me the opportunity to apply for an internship at their esteemed and forward-thinking organization. Their guidance and encouragement have played a crucial role in my development of this scalable LLM chatbot system.

This project has given me the chance to demonstrate my abilities and knowledge while also enabling me to delve into cutting-edge AI technologies, grasp production methods, and make a valuable contribution. I am grateful for the time and effort put into developing this assignment and am eager to continue learning and advancing with Sarvam AI.

**Table of Content**

|  |  |  |
| --- | --- | --- |
| Sl.No. | Title | Page No. |
| 1. | Problem Statement | 4 |
| 2. | System Overview | 5 - 7 |
| 3. | System Architecture | 8 - 10 |
| 4. | Scalability | 11 |
| 5. | Cost Considerations | 12 - 13 |
| 6. | LLM Integration | 14 - 15 |
| 7. | Frontend Implementation | 16 |
| 8. | Dockerization and Kubernetes | 17 |
| 9. | Deployment Instructions | 18 - 19 |

**PROBLEM-STATEMENT**

Build a Scalable LLM Chatbot System Overview Design and implement a chat application powered by a Large Language Model (e.g., OpenAI, Anthropic, Llama, etc.) that can support 10,000+ users. Your solution should demonstrate your approach to building production-ready systems.

**System Overview**

Our chatbot system is built to accommodate more than 10,000 users by utilizing a Large Language Model (LLM) for engaging conversations. It incorporates the Gemini API for client-side functionalities and uses React for the user interface. To ensure scalability and reliability, we implement Docker and Kubernetes for containerization and orchestration.

1. **Core Functionality:**

The main capabilities of our scalable LLM chatbot system consist of various essential features aimed at delivering users a smooth and interactive conversation experience.

1. ***Chat Interface:***

* **User-Interaction:** Users have the ability to send messages and participate in discussions with the LLM.
* **Real-Time Updates:** The chat log refreshes instantly whenever new messages are sent or received.
* **Responsive Design:** Guarantees a seamless user experience on various devices.

1. ***Conversation History:***

* **Contextual Understanding:** Keeps a log of past interactions to deliver responses that are contextually appropriate.
* **Data Storage:** The conversation history is saved in a MongoDB database to allow for quick access.

1. ***Scalability:***

* **Support for Large User Base:** Built to accommodate many users at the same time.
* **Horizontal-Scaling:** Kubernetes makes it simple to add more pods to manage higher traffic levels.
* **Vertical-Scaling:** Performance can be enhanced by increasing the resources (CPU, RAM) allocated to current pods.

1. **LLM Integration:**

* **Gemini-Api:** Makes use of the Gemini API to engage with the LLM.
* **Prompt Engineering:** Custom prompts are designed to improve the quality of responses.
* **Fine-Tuning:** The model's parameters are modified to enhance its performance.

1. **Containerization with Docker:**

* **Consistent Environment:** Docker guarantees that the frontend and backend applications operate in uniform environments on various machines.
* **Efficient Deployment:** Docker images are efficient and simple to deploy, minimizing the time and resources required for installation.
* **Isolation:** Docker containers offer separation between applications, enhancing security and minimizing conflicts.

1. **Orchestration with Kubernetes:**

* **Automated Deployment:** Kubernetes streamlines the deployment and scaling processes for Docker containers.
* **Self-Healing:** Kubernetes guarantees that if a container crashes, it will be restarted automatically.
* **Resource Management:** Kubernetes enhances the distribution of resources to guarantee the effective utilization of computing assets.

**EXAMPLE INTERACTION:**

**User:** explain theoretical physics to a toddler

**AI**: Imagine you have LEGOs! Theoretical physics is like building the biggest, coolest LEGO castle ever, but *without* actually having all the LEGOs.

We know some LEGOs exist (like the ones we see around us – that's like the stuff we can *see* and *touch* in the world). But theoretical physicists try to figure out what *other* LEGOs might be out there, even if we can't see them yet. They imagine tiny, tiny LEGOs that make up everything, or maybe even LEGOs that are *bigger* than the whole universe!

They use their imaginations and clever math to build their castle, and then they test their ideas to see if they fit with what we *do* know about the real LEGOs (the things we can see). Sometimes they're right, and sometimes they need to rebuild parts of their castle!

So, theoretical physics is like a giant LEGO castle made of *ideas* about how the universe works, and scientists keep building and changing it to make it even better. It's all about asking big questions like: "What is everything made of?" and "How did the universe begin?" It's super fun, even if it's a bit tricky!

**Summary:**

This essential feature establishes a strong basis for a reliable and user-friendly chatbot system capable of managing a significant volume of users while ensuring optimal performance and dependability. Utilizing Docker and Kubernetes facilitates effective deployment and scalability.

**System Architecture**

The system architecture is built to accommodate a significant number of users while ensuring optimal performance and dependability. It includes several essential elements:

1. **Frontend (React)**

* **User Interface**: Manages user interactions and presents chat dialogues.
* **Real-time Updates**: Refreshes the chat log instantly as new messages come in or are sent.
* **Responsive Design**: Ensures a smooth user experience across various devices.

1. **Backend (Node.js)**

* **API Management**: Handles API requests, engages with the LLM, and keeps track of conversation history.
* **LLM Integration**: Makes use of the Gemini API to engage with the LLM.
* **Database Interaction**: Saves and accesses conversation history using MongoDB.

1. **LLM Integration (Gemini API)**

* **Text Generation**: Utilizes the Gemini API to create responses according to user input, which can include both text and images.
* **Prompt Engineering**: Custom prompts are designed to improve the quality of responses.
* **Fine-Tuning**: Model parameters are fine-tuned to enhance performance.

1. **Database (MongoDB)**

* **Conversation History**: Keeps a record of conversation history for context.
* **Data Retrieval**: Retrieves relevant data to inform responses.

1. **Containerization (Docker)**

* **Consistent Environments**: Docker guarantees that the frontend and backend applications operate in uniform environments on various machines.
* **Efficient Deployment**: Docker images are efficient and simple to deploy, which minimizes the time and resources required for installation.

1. **Orchestration (Kubernetes)**

* **Scalability**: Kubernetes makes it simple to add more pods to manage higher traffic levels.
* **Self-Healing**: Guarantees that a container will be automatically restarted in the event of a failure.
* **Resource Management**: Enhances the distribution of resources to guarantee effective utilization of computing assets.

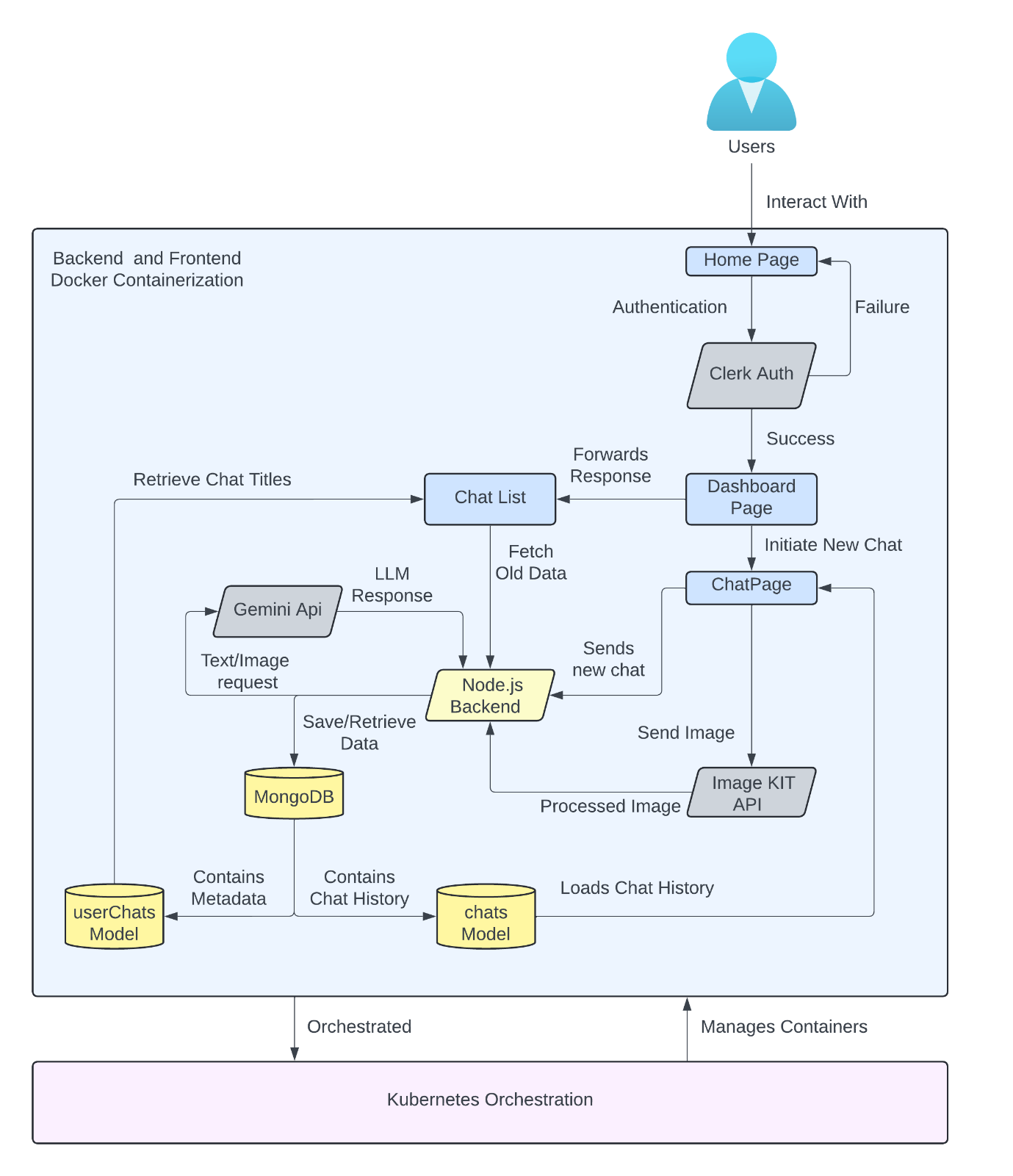
1. **API Gateway**

* **Unified Entry Point**: Offers a centralized access point for API requests.
* **Traffic Management**: Effectively handles API traffic to guarantee seamless functionality.

1. **Monitoring and Logging**

* **Performance Monitoring**: Monitors system performance to detect any bottlenecks.
* **Error Logging**: Records errors for troubleshooting and enhancement.

**System Architecture Diagram:**

****

**Fig.1: System-Architecture for the Proposed LLM ChatBot System**

**Scalability**

Scalability is essential for systems that must accommodate a rising number of users or higher traffic levels. Our system is built to scale both horizontally and vertically, allowing it to effectively handle increased demand while maintaining performance. Important measures implemented to enhance the program's scalability include Horizontal Scaling (HPA Autoscaler), Vertical Scaling, and testing the Service and Load Balancer Configuration.

**Horizontal (HPA Autoscaler) Scaling**

* Configured Kubernetes Horizontal Pod Autoscaler (HPA) to automatically adjust the number of pods according to CPU and memory usage.
* This enables the system to automatically assign additional pods during periods of increased traffic.

Challenge: Constraints of the Host System (insufficient memory and processing capabilities)

**Vertical Scaling**

* Enhanced performance by raising CPU and memory limits for current pods.
* It allows each pod to manage a greater volume of requests before additional ones are generated.

Challenge: If one pod hits its resource limits, it may still create a bottleneck.

**Service & Load Balancer Configuration**

* Created a Kubernetes Service to make the backend accessible through a NodePort service.

Challenge: As we were conducting tests locally on Minikube, we didn't have access to a proper LoadBalancer. In a genuine cloud setting (such as AWS, GCP, or Azure), we would be able to utilize an external LoadBalancer to effectively manage traffic distribution.

**Possible Bottlenecks:**

MongoDB Atlas can lead to complexities in replication and sharding, making rebalancing more challenging during scaling operations. Additionally, Clerk, the authentication tool, may experience reliability problems when scaled, such as downtime and sluggish API response times.

While this method allows for some scalability, additional improvements and testing are required for the cloud environment. On the other hand, utilizing Kubernetes enhances reliability through its failure management features, including pod replication and automatic restarts in the event of failures.

**Cost Considerations**

Although this project is currently offered at no charge, significant funding will be necessary if it moves into production. These funds will mainly be needed for development, infrastructure, and maintenance. To help reduce these expenses, we can:

* Emphasize Modular Development
* Implement a Pay-As-You-Go Model
* Utilize Cloud Services that provide scalable infrastructure solutions
* Employ Kubernetes for orchestrating

**LLM Integration**

The project utilized the Google Gemini-1.5-flash API to integrate a large language model (LLM). This version of Gemini offers quick and affordable responses, making it suitable for real-time use. Additionally, safety protocols were implemented, covering areas such as harassment and hate speech.

**Prompt Engineering and Fine-Tuning:**

To achieve reliable outcomes, a comprehensive system prompt was created that provides both a creative and factual perspective on the question. The main elements of the prompt consist of:

* Technical Explanations
* Scientific Analysis
* Creative Writing
* Step-by-Step guides
* Compartitive Analysis
* Concise Summaries
* Historical or Contextual Backgrounds
* Hypothetical Scenarios
* Conversational and Casual Responses
* Problem Solving and Troubleshooting
* Challenging assumptions
* Speculative and Futuristic Thinking
* Creative Thinking
* Ethical and Philosophical Explorations

In addition, specific directives were provided to:

* Avoid pseudoscience and provide precise and correct facts when discussing scientific topics.
* Use of First-principles-thinking concept when solving a problem.
* Considering both Traditional and Unconventional Responses for a question.

In addition to Prompt Engineering, the model underwent Fine Tuning. A temperature setting of 0.5 was used to strike a balance between creativity and factual accuracy. A Nucleus sampling rate (top\_p) of 0.9 was implemented to produce more controlled responses, while a top-K Sampling (top\_k) rate of 40 was applied to allow the model to choose the next token from the 40 most likely words, enhancing the model's diversity.

**Challenges Faced:** Determining the values of the parameters.

**Solution:** Constant Testing

**Frontend Implementation**

The entire frontend was developed using the React 19 library because of its flexibility, efficiency, and lightweight nature.

All the pages are located in the src/routes directory, the layout files are in src/layouts, and the components can be found in src/components. Each of these directories has a specific role in the project: the routes folder manages the pages for each route, the layouts folder contains the Dashboard and Root layouts, and the components folder is responsible for functionalities like New Prompt, Chat List, and Upload Picture.

**Dockerization and Kubernetes**

To prepare the project for production, we implemented Dockerization and Kubernetes. By dockerizing the files, we package all dependencies and runtime settings into a portable Docker image. This image is utilized in the Kubernetes manifests, which oversee the deployment, scaling, and networking of the application. The configuration includes a ConfigMap for environment settings, a Deployment that executes the Docker image, a Horizontal Pod Autoscaler (HPA) to handle scaling based on CPU usage, and a Service that makes the application accessible externally.

**Challenges Faced:**

As someone who is new to Kubernetes and Docker, I encountered several challenges, such as:

* Missing environment variables in Docker
* Incorrect configurations in the DockerFile
* Setting up Kubernetes
* Stress testing due to system limitations

Despite having these issues this project helped a lot in learning on how to make a Production-Ready System.

**Deployment Instructions**

Firstly Clone the project using

`**gh** repo clone PiyushKBhattacharyya/LLM-ChatBot` and procced with the following steps for deployment:

* **Deployment without Docker:**

To Deploy the project on your localhost ; without Docker, follow the following steps:

1. Setup Environment Variables:
   1. Backend:
      1. Navigate to the backend using `cd backend` command in the terminal
      2. Create a file named ‘.env’ which contains:

IMAGE\_KIT\_ENDPOINT=<YOUR\_IMAGE\_KIT\_ENDPOINT\_URL>

IMAGE\_KIT\_PUBLIC\_KEY=<YOUR\_IMAGE\_KIT\_PUBLIC\_KEY>

IMAGE\_KIT\_PRIVATE\_KEY=<YOUR\_IMAGE\_KIT\_PRIVATE\_KEY>

CLIENT\_URL=http://localhost:5173

MONGO=<YOUR\_MONGO\_DB\_CONNECTION\_STRING>

CLERK\_PUBLISHABLE\_KEY=<YOUR\_CLERK\_PUBLISHABLE\_KEY>

CLERK\_SECRET\_KEY=<YOUR\_CLERK\_SECRET\_KEY>

PORT=3000

* 1. Frontend:
     1. Navigate to the client folder using `cd client` command in the terminal.
     2. Create a file named ‘.env’ which contains:

VITE\_CLERK\_PUBLISHABLE\_KEY=<YOUR\_CLERK\_PUBLISHABLE\_KEY>

VITE\_IMAGE\_KIT\_ENDPOINT=<YOUR\_IMAGE\_KIT\_ENDPOINT\_URL>

VITE\_IMAGE\_KIT\_PUBLIC\_KEY=<YOUR\_IMAGE\_KIT\_PUBLIC\_KEY>

VITE\_GEMINI\_PUBLIC\_KEY=<YOUR\_GEMINI\_API\_PUBLIC\_KEY>

VITE\_API\_URL=http://localhost:3000

1. Install Dependencies:
   1. Navigate to the Backend Folder using `cd backend` command in the terminal and use ‘npm i –force’ command to install dependencies
   2. Navigate to the Frontend Folder using `cd client` command in the terminal and use ‘npm i –force’ command to install dependencies
2. Run the Backend Server:
   1. Navigate to the Backend Folder using `cd backend` command in the terminal
   2. Start the backend Server by using `npm start`
3. Run the Client App:
   1. Navigate to the Backend Folder using `cd client` command in the terminal
   2. Start the client App by using `npm run dev`

Your App now should be running at Localhost:5137 with the backend at Localhost:3000

* **Deployment using Docker:**

To deploy the project using docker, follow the following steps:

1. Setup Environment Variables (Follow Step 1 from Deployment without using Docker)
2. Build and run the container using `docker-compose up --build `