SARVAM AI:

HOME-ASSIGNMENT DOCUMENTATION

**Acknowledgement**

I would like to express my sincere gratitude to Saurabh Karn, Founding Member of Sarvam AI, and Nitesh Methani, Machine Learning Engineer at Sarvam AI, for providing me with the opportunity to complete this home assignment as part of the internship application process. Their guidance and support have been invaluable in helping me develop this scalable LLM chatbot system.

This assignment has not only allowed me to demonstrate my skills and knowledge but also given me a chance to explore innovative AI technologies and contribute to a meaningful project. I appreciate the time and effort invested in creating this assignment and look forward to the opportunity to learn and grow further 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 designed to support over 10,000 users by leveraging a Large Language Model (LLM) for conversational interactions. The system integrates the Gemini API for client-side operations and utilizes React for the frontend. Docker and Kubernetes are employed for containerization and orchestration, ensuring scalability and reliability.

1. **Core Functionality:**

The core functionality of our scalable LLM chatbot system includes several key features designed to provide users with a seamless and engaging conversational experience.

1. ***Chat Interface:***

* **User-Interaction:** Users can input messages and engage in conversations with the LLM.
* **Real-Time Updates:** The chat log updates in real-time as new messages are sent or received.
* **Responsive Design:** Ensures a smooth user experience across different devices.

1. ***Conversation History:***

* **Contextual Understanding:** Maintains a record of previous interactions to provide contextually relevant responses.
* **Data Storage:** Conversation history is stored in a MongoDB database for efficient retrieval.

1. ***Scalability:***

* **Support for Large User Base:** Designed to handle over 10,000 users simultaneously.
* **Horizontal-Scaling:** Kubernetes allows for easy addition of more pods to handle increased traffic.
* **Vertical-Scaling:** Resources (CPU, RAM) can be increased for existing pods to improve performance.

1. **LLM Integration:**

* **Gemini-Api:** Utilizes the Gemini API for interacting with the LLM.
* **Prompt Engineering:** Custom prompts are crafted to enhance response quality.
* **Fine-Tuning:** Model parameters are adjusted for better performance.

1. **Containerization with Docker:**

* **Consistent Environment:** Docker ensures that both the frontend and backend applications run in consistent environments across different machines.
* **Efficient Deployment:** Docker images are lightweight and easy to deploy, reducing the time and resources needed for setup.
* **Isolation:** Docker containers provide isolation between applications, improving security and reducing conflicts.

1. **Orchestration with Kubernetes:**

* **Automated Deployment:** Kubernetes automates the deployment and scaling of Docker containers.
* **Self-Healing:** Kubernetes ensures that if a container fails, it will be automatically restarted.
* **Resource Management:** Kubernetes optimizes resource allocation to ensure efficient use of computing resources.

**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 core functionality provides a solid foundation for a robust and user-friendly chatbot system that can handle a large number of users while maintaining high performance and reliability. The use of Docker and Kubernetes ensures efficient deployment and scalability.

**System Architecture**

The system architecture is designed to support a large number of users while maintaining high performance and reliability. It consists of several key components:

1. **Frontend (React)**

* **User Interface**: Handles user interactions and displays chat conversations.
* **Real-time Updates**: Updates the chat log in real-time as new messages are sent or received.
* **Responsive Design**: Ensures a smooth user experience across different devices.

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

* **API Management**: Manages API requests, interacts with the LLM, and maintains conversation history.
* **LLM Integration**: Utilizes the Gemini API for interacting with the LLM.
* **Database Interaction**: Stores and retrieves conversation history from MongoDB.

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

* **Text Generation**: Uses the Gemini API to generate responses based on user input (both text and images).
* **Prompt Engineering**: Custom prompts are crafted to enhance response quality.
* **Fine-Tuning**: Model parameters are adjusted for better performance.

1. **Database (MongoDB)**

* **Conversation History**: Stores conversation history for context.
* **Data Retrieval**: Retrieves relevant data to inform responses.

1. **Containerization (Docker)**

* **Consistent Environments**: Docker ensures that both the frontend and backend applications run in consistent environments across different machines.
* **Efficient Deployment**: Docker images are lightweight and easy to deploy, reducing the time and resources needed for setup.

1. **Orchestration (Kubernetes)**

* **Scalability**: Kubernetes allows for easy addition of more pods to handle increased traffic.
* **Self-Healing**: Ensures that if a container fails, it will be automatically restarted.
* **Resource Management**: Optimizes resource allocation to ensure efficient use of computing resources.

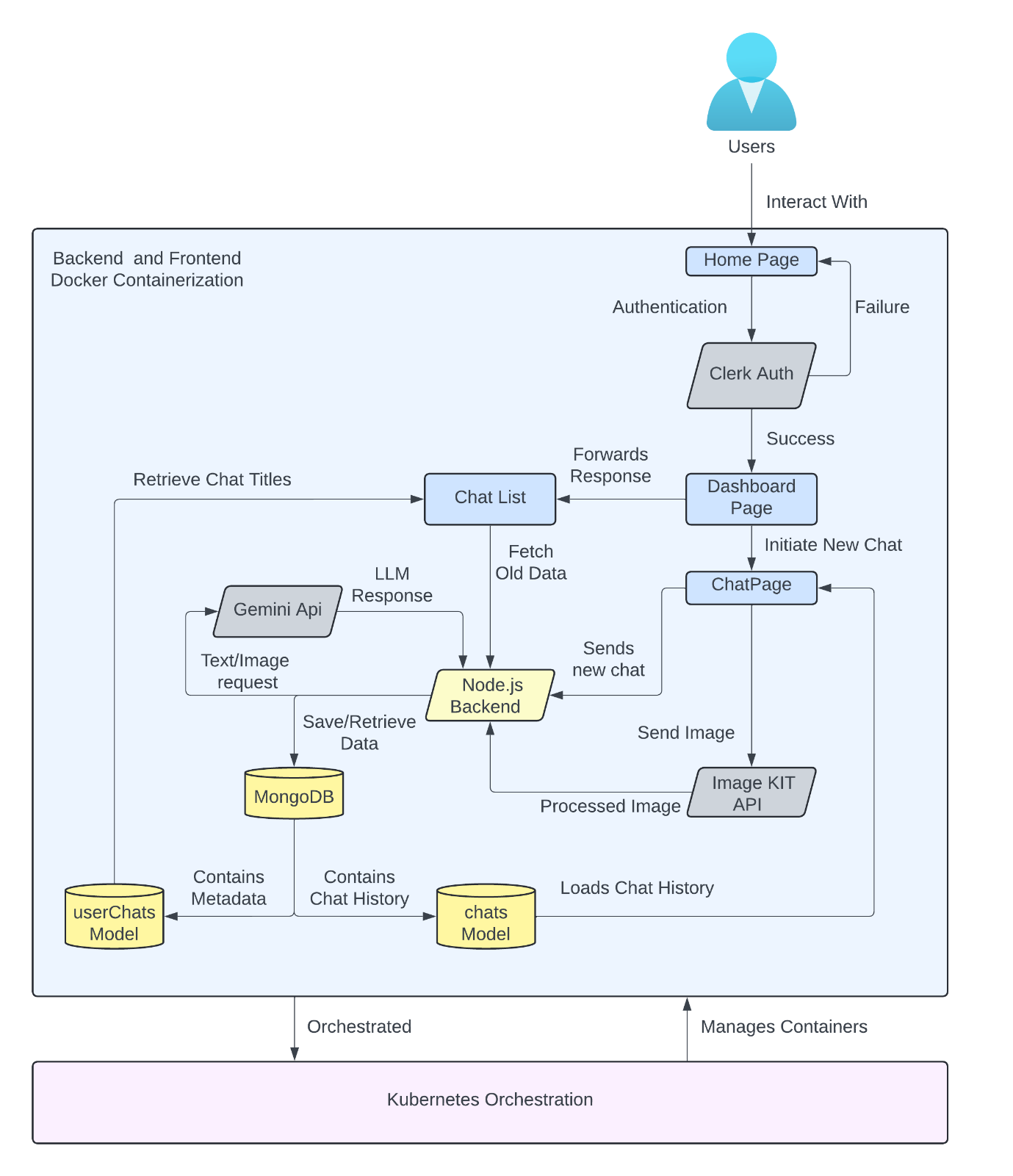
1. **API Gateway**

* **Unified Entry Point**: Provides a single entry point for API requests.
* **Traffic Management**: Manages API traffic efficiently, ensuring smooth operation.

1. **Monitoring and Logging**

* **Performance Monitoring**: Tracks system performance to identify bottlenecks.
* **Error Logging**: Logs errors for debugging and improvement.

**System Architecture Diagram:**

****

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

**Scalability**

Scalability is crucial for systems that need to handle a growing number of users or increased traffic. Our system is designed to scale both horizontally and vertically, ensuring that it can efficiently manage increased load without compromising performance. Key steps taken to scale the program includes Horizontal (HPA Autoscaler) Scaling, Vertical Scaling and test Service and Load Balancer Configuration

**Horizontal (HPA Autoscaler) Scaling**

* Configured Kubernetes HPA to automatically scale pods based on CPU and memory utilization.
* This allows the system to dynamically allocate more pods when traffic spikes.

Challenge: Finding the right CPU/memory thresholds to balance cost and performance.

**Vertical Scaling**

* Increased CPU and memory limits for existing pods to enhance performance.
* This ensures each pod can handle a higher number of requests before new ones are created.

Challenge: If a single pod reaches its resource limits, it could still become a bottleneck.

**Service & Load Balancer Configuration**

* Created a Kubernetes Service to expose the backend using a NodePort service.

Challenge: Since we were testing locally on Minikube, a proper LoadBalancer was unavailable. In a real cloud environment (AWS, GCP, Azure), we could use an external LoadBalancer to distribute traffic efficiently.

**Possible Bottlenecks:**

MongoDB Atlas can cause Replication and Shrading Complexity which can complicate rebalancing during Scaling Operations and Clerk which is used for authentication can run into reliability issues when scaled which includes downtime and slow API Response time.

This approach makes the program scalable to a extent, but further enhancements and testing needs to be done for Cloud Environment. However, the use of Kubernetes also ensures reliability due to failure handling properties by pod replication and automatic restarts in case of failures

**Cost Considerations**

While this project is made free of cost, if led to production will need fundings on a large scale. They would primarily be required for Development, Infrastructure as well as maintainance.

To mitigate these costs; we can:

* Focus on Modular Development
* Incorporate Pay-As-You-Go Model
* Leverage Cloud Services that offer scalable infrastructure options
* Use Kubernetes for orchestrating

**LLM Integration**

Google gemini-1.5-flash API was used to incorporate LLM into the project. Gemini-1.5-flash provides fast and cost-effective responses, making it ideal for real-time applications. Safety measures were also incorporated with categories including HARM\_CATEGORY\_HARASSMENT and HARM\_CATEGORY\_HATE\_SPEECH

**Prompt Engineering and Fine-Tuning:**

To ensure consistent results, a detailed system prompt was designed giving it a creative as well as a factual outlook for the question. Key features of the prompt includes:

* 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

Furthermore, special instructions were given 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.

Fine Tuning was also done to the model in addition to Prompt Engineering; a temperature of 0.5 was implied for a balance between creative and factual responses, a Nucleus sampling rate (top\_p) of 0.9 was applied to generate more controlled response, a top-K Sampling (top\_k) rate of 40 was applied to make the model to select the next token from top 40 most probable words making the model more diverse.

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

**Solution:** Constant Testing

**Frontend Implementation**

The React 19 library was used to produce the entirety of frontend, due to its versatility, performance and lightweight capabilities.

All the pages are inside the src/routes folder, the layout in src/layouts and components in src/components. Each of these folders serve a purpose in this project; routes folder handles the pages of each route, layouts handle the Dashboard and Root layouts of the project whereas components handle the New Prompt, Chat List and Upload Picture functionalities.

**Dockerization and Kubernetes**

To make the project Production Ready, Dockerization and Kubernetes were applied. By dockerizing your files, we encapsulate all dependencies and runtime configurations into a portable Docker image. This image is then referenced by your Kubernetes manifests, which manage the application’s deployment, scaling, and networking. The setup includes a ConfigMap for environment configuration, a Deployment that runs the Docker image, a Horizontal Pod Autoscaler (HPA) to manage scaling based on CPU utilization, and a Service that exposes the application externally.

**Challenges Faced:**

Due to being new to Kubernetes and Docker, I personally faced a lot of challenges which included:

* Missing environment variables in Docker
* Incorrect DockerFile setups
* Kubernetes Setup
* Stress Testing (due to System Constraints)

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

**Deployment Instructions**

* **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:
   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. Client:
   * 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. Build and run the container using `docker-compose up --build `