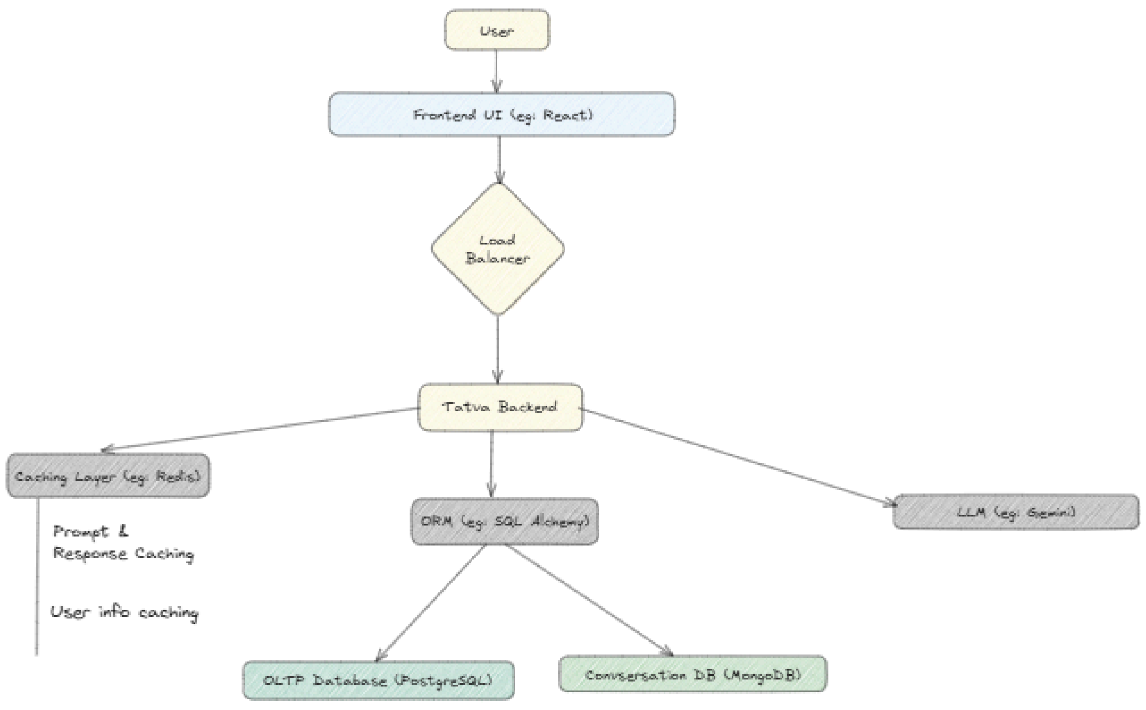
Overview

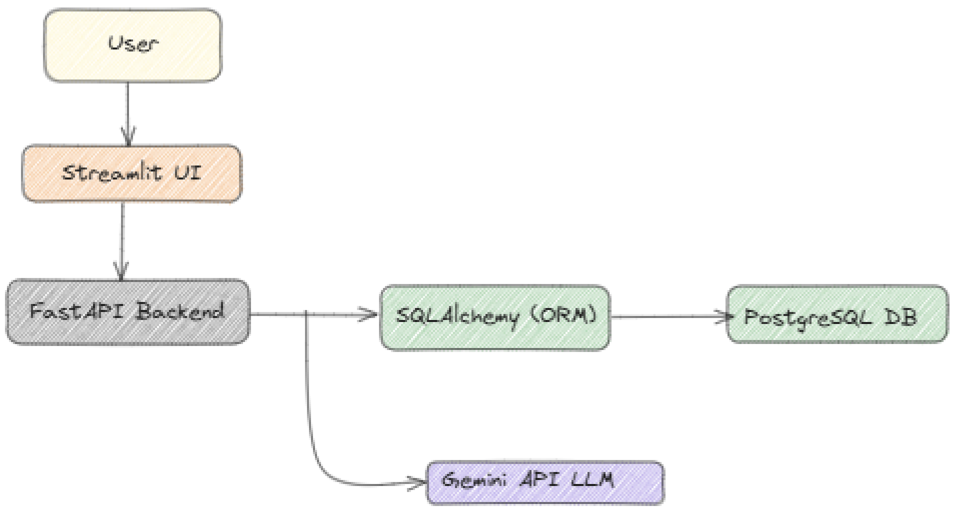
Tatva AI (in sanskrit translates to Essence) - a scalable, reliable, and cost-effective chat application powered by a Large Language Model (LLM), designed to handle 10,000+ users simultaneously. It includes frontend, backend, database management, and advanced AI features to ensure a smooth user experience and strong system performance. The solution assumes an external LLM provider is available. Whether this provider is a cloud-based service or a custom-trained model can change in the future based on needs.

High-Level System Diagram of Tatva AI



The above diagram is how I see to implement the chatbot. Given the time constraint, I have done a simplified version of the proposed solution.

Tatva AI Implementation Diagram



Components and Interaction

- Tatva AI's frontend UI is built using Streamlit, providing an easy-to-use interface suitable for quick prototyping. and dependencies required for the project.
- Load Balancer: Load balancer should be utilized to distribute user requests to handle traffic.
- Tatva AI's backend leverages FastAPI which manages asynchronous requests, manages user login (user authentication), sending messages, fetching previous conversations and integration with databases..
- SQLAlchemy (ORM): Using this for database operations, simplifying database interactions.
- Using PostgreSQL DB as it is efficient for quick DB transactions and fast querying. The below table displays the schema followed:

Tables	Schema
Users	- id (Integer, Primary Key) - username (String, Unique) - email (String, Unique) - hashed_password(String)
Conversations	- id (UUID String, Primary Key) - user_id (Integer, Foreign Key -> users.id) - title (String) - created_at (DateTime)
Messages	- id (String UUID, Primary Key) - conversation_id (String, Foreign Key -> Conversations) - sender (String, User/Bot) - text (String) - timestamp (DateTime)

- Using Gemini API that delivers prompt responses.

Scalability Considerations for Tatva AI

Apart from using a load balancer to distribute the requests across instances, we can do the following:

- PostgreSQL Optimization: PostgreSQL should have indexing on specific fields for faster queries. PostgreSQL scales well with indexing. Attributes like conversation.id, user.id etc, should be indexed.
- Asynchronous Backend: Utilize FastAPI's asynchronous processing capabilities to stream responses efficiently.
- Redis Caching: Employ caching of frequent queries and responses to lower database loads and reduce response times.

Reliability

While the demo is implemented and tested locally, there is a need to ensure reliability for the software to be good. The below points can be used to address the above.

Fault Tolerance

- Deploy system components using Kubernetes clusters for redundancy of infrastructure, recovery in case of failures and auto-scaling during load spikes.
- Employ PostgreSQL replication to account for regional failures.
- Implement automated system health checks and real-time alerts for immediate identification of failures.

Accounting for dooms-day scenario

Loss of data can result in a dooms-day scenario which can be resolved as given below

- Ensure periodic database snapshots to help in reliable data recovery.
- Do regular chaos testing to simulate such failure modes.

Cost Considerations

Efficiency in LLM-Based Systems

- Implement Redis caching extensively to minimize redundant LLM API calls.
- Leverage Prompt Caching techniques that are now being supported by various LLM providers.

LLM Integration

Integration Strategy

I have utilized Gemini for my implementation, but the LLM provider can be switched later to OpenAI / Anthropic with few changes as highlighted below.

- Library/API adjustments:
  Rewiring backend services using different libraries and API endpoints specific to OpenAI or Anthropic, requiring minor code adjustments primarily within the API interaction layer.

- Prompt Engineering and System Tuning:
  Adapt prompt engineering methods and fine-tune system responses to align with the nuances of the new provider.

- Perform A/B testing during the provider transition to identify the best set of prompts and configurations.

Context Management and Prompt Engineering

- Maintain conversational histories to ensure the LLM has the necessary context.
- Improve prompts through prompt optimization techniques if necessary.

Advanced Techniques

- RAG based features

  Use Retrieval-Augmented Generation (RAG) to provide accurate responses by pulling information from external knowledge bases. For example, users can upload PDFs or documents, and the chatbot can directly answer questions based on the content of those files.

  (Note: Fine-tuning can be considered separately for more specialized use cases.)

- Handling Model Failures

  To ensure a reliable experience in cases of an LLM provider going down, it makes sense for Tatva to rely on multiple LLM providers and follow a circuit breaker pattern, to switch between providers.

- Personalization in Tatva

  - Personalizing large language models (LLMs) make interactions more user friendly and effective. Some methods are suggested below

- Rather than have a static/ generic system prompt (used a very basic version in the implementation), using conditional system prompting is better as it engages in dynamic adjustments based on the user interactions.

- This can be extended by synthesizing memory for the model based on what the user chats about. This could be injected in the system prompt itself whenever required.

- RL based pipelines can be introduced to allow the model to continually learn the model from user feedback. The system (prompts as well as model) can be improved through user feedback by leveraging RL based on the user feedback and training more specialized/ personalized models.