GAIA Application – High Level Design

**Description and General Requirements**

1. GAIA is a software based solution for implementing open source large language models adapted to the specific user needs of the aerospace engineer.
2. GAIA provides the capability to select a specific model that was fine-tuned on relevant data and use it to answer questions and generate content from user provided documents.
3. The software is divided into several micro-services. Each of them is run as a containerized application. These services are:
   1. **webUI service** – a React web application that generates the user interface for the system.
   2. **GAIA Server** – a python backend server that handles all user requests and performs the necessary actions by communicating with additional resources.
   3. **Inference** – a service that loads LLM models to the GPU, and performs inference on text queries received via HTTP.
   4. **Monitoring** – a service that provides information on the health of the system via a web based dashboard.
4. GAIA shall be built to handle up to 1000 concurrent users. This shall be done by using a scalable architecture.
5. webUI (**WUI**) requirements:
   1. GAIA shall enable the user to select the currently active LLM model
   2. GAIA shall enable to user to chat with the LLM model. All chat history shall be stored and the user shall be able to resume previous chats.
   3. GAIA shall allow the user to rate the quality of the answer received from the model.
   4. GAIA shall include personalized behavioral features – showing the user’s photo, greeting the user and asking about previous chats, suggesting follow up questions after each question.
   5. GAIA shall enable the user to select from a set of pre-embedded vector stores based on large text corpus for performing Retrieval Augmented Generation (RAG) where the answer to a query contains data augmented from a large text corpus.
   6. GAIA shall enable the user to load a single text document and ask questions / perform text generation using the document. For this purpose GAIA shall temporarily create a vector DB embedding of the document.
   7. GAIA shall enable the user to send the entire chat or a specific answer to an e-mail address. The button shall open an outlook window with a cover letter and the text pasted in.
6. GAIA Server (**GS**) requirements:
   1. GAIA shall provide an HTTP API for the webUI and perform the necessary actions after each command from the user.
   2. GAIA shall connect to an LDAP authentication service and retrieve the user groups for each connection
   3. GAIA shall allow access only to specific models according to user groups
   4. GAIA shall allow access only to specific vector embeddings according to user groups
   5. GAIA shall store user interactions in a private location per user (P:\ drive)
   6. GAIA shall send monitoring data to an OTLP endpoint for display by the monitoring dashboard.
7. Text Generation Inference (TGI) requirements:
   1. The container shall Load specific model files upon startup and provide an access port for each model.
   2. The container shall allow changing the currently loaded model endpoints.
   3. Distribution of load between available GPUs
   4. Report monitoring data to the monitoring server (MS)
8. Monitoring Server (MS) requirements:
   1. Collect data on system health from GS and TGI servers
   2. Display a dashboard with current health using Grafana

**Interfaces**

**WUI to GS interface**

This interface provides the main backbone of the application as it allows the web UI to control all of the functions of the server. The interface is based on HTTP requests from the React frontend to the python streamlit backend, at port 5050.

1. User API:
   * **Endpoint**: /api/user
   * **Description**: This API handles user authentication and authorization.
   * **Endpoints**:
     + POST /api/user/login: Authenticates a user and generates an authentication token. This has to be called first.
     + GET /api/user/get\_models: gets the model list allowed for the user
     + GET /api/user/get\_embeddings : gets the embedding list
     + POST /api/user/get\_welcome : passes the most recent chat text and gets a welcome message for the user generated by the LLM
     + POST /api/user/select\_model : passes the selected model that the user wants to chat with
     + POST /api/user/creativity : passes the required creativity level for the model creative / balanced / accurate
2. Chat API:
   * **Endpoint**: /api/chat
   * **Description**: This API handles user interactions and model responses in the chat interface.
   * **Endpoints**:
     + POST /api/chat/message: Receives a user message and returns the model's response.
     + POST /api/chat/new: clear the current chat context and start a new chat.
     + POST /api/chat/resume : sends the chat history for a resumed (older) chat.
     + POST /api/chat/feedback: Allows users to provide feedback on the quality of the last response. The user sends a score of 0 to 3.
     + GET /api/chat/title : gets a title for the current active chat
     + GET /api/chat/followup: Retrieves a list of follow up questions.
3. Document API:
   * **Endpoint**: /api/docs
   * **Description**: This API handles uploading and processing of single PDF or text documents.
   * **Endpoints**:
     + POST /api/docs/upload: Uploads a single PDF or text document to the server.
     + POST /api/docs/clear: clears the current loaded file embedding from memory.
4. Embedding Set API:
   * **Endpoint**: /api/embedding-sets
   * **Description**: This API manages embedding sets for the model.
   * **Endpoints**:
     + GET /api/embedding-sets: Retrieves available embedding sets.
     + POST /api/embedding-sets/select: Selects an embedding set for the user. Passes the name of the set. If an empty name is passed then embedding is not used.