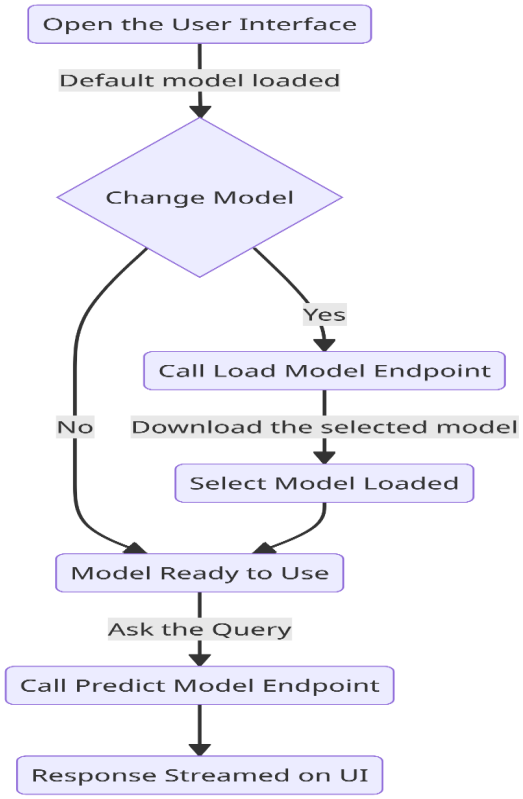
Objective

To create a chatbot using a containerized Large Language Model (LLM) and deployed as a web application for user interaction. The containerized LLM accepts user queries and provides relevant responses in real-time.

Solution Overview

This project involves containerizing a Large Language Model using Docker and deploying it as a web application for chatbot interaction. The high-level steps to accomplish this are as follows:

- Creating a front-end for chatbot, using which the user can enter its query and see the chatbot's response. Additionally, the user can choose a specific LLM of its choice. Selecting an appropriate LLM keeping resource constraints and latency of response in mind.
- Providing the LLM a concise and apt prompt.
- Creating a backend API, which is used to query and fetch the response from the LLM.
- Streaming the response to make the chat-bot more interactive.



Solution Architecture

Installation

To get started with the chatbot install **`Python`** and **`Python Package Index`**.

The solution requires **`docker`** and **`docker-compose`**.

*Note: For development we have used **`python==3.9,`** but using any version of python above will not cause any issues*

Once the tools are installed. We need to install the dependencies for running the Gradio web-app.

First step is to set up a virtual environment. For this setup the following command.

```
python3 -m venv chatbot_ui_env
```

To activate the virtual environment, use the following command:

```
source chatbot_ui_env/bin/activate
```

To install the dependencies by following the following command:

```
pip install -r requirements.txt
```

Installing without Docker

In case **`Docker`** is not available in the system. Following steps can be done to get the necessary requirements,

Go to the chatbot directory in the repo using the command.

```
cd chatbot/
```

Install the required dependencies using the command.

```
pip install -r requirements.txt
```

Steps to run the application

Once the dependencies are installed, the application can be started using the build_run.sh script.

Command to run the script::

```
bash build_run.sh
```

The script above performs the following steps:

- Download the default model into the model directory.
- Create the docker image.
- Run the docker image.
- Run the web application.

.Steps to run the application without docker

To run the application without docker, the backend and frontend needs to be run separately.

1. Steps to start the backend API:
   1.1. Navigate to the `src` dir in chatbot.
   ```
   cd chatbot/src/
   ```
   1.2. Run the FastAPI using uvicorn:
   ```
   uvicorn main:app –port 8100:8100 –reload
   ```
2. Steps to start the frontend app
   2.1. Navigate to the `web` dir in the repo
   ```
   cd web/
   ```
   2.2. Start the `gradio` application.
   ```
   gradio app.py
   ```

Open the URL **http://127.0.0.1:8200/** on the browser to access the UI.

Modules

The solution has two key modules - front-end and backend. Both of them have been explained below.

Front-end

- The front-end user interface is built on python, using the `gradio` framework.
- The UI has 4 key components:
    - **Model status tab**: It shows the status of the model, when we change the model, it indicates what steps are being performed and when the model is ready to be used.
    - **Model Selection tab**: A section that displays all the LLMs that the system supports, the user can select the radio button of whichever model he wants and the model will download/load on the backend. In case none of the models is selected, the default choice is `GPT-J`.
    - **Chat section**: This area contains the text bar and the view of the previous interactions between user and bot.
    - **Clear button**: To clear the existing conversation and start a new one.
- Front-end URL can be accessed using the following link **`http://127.0.0.1:8200/`**

Backend

- The backend of the system is written in **python** and is made in an API using the package **FastAPI**. The backend system is containerized using **docker**.
- The backend API has two endpoints.
    - **Model Loading endpoint**: The endpoint takes the model name as input and downloads, loads the model to be used.
    - **Model Prediction endpoint**: The endpoint takes the user query as input and returns the prediction of the language model.

- We create a prompt to input in the model, to control the response we get the model. We can experiment with the prompt based on the use-case.
- There are a bunch of error conditions we have identified in the system, that gives us an informative response from the API incase of the error and makes it easier for the developer to debug.

API Request and Response

Model Loading endpoint:

- **Endpoint:** http://127.0.0.1:8100/load-model/
- **Request**

```
{
  "model_name": "LLAMA"
}
```

- **Response:** `None`

Model Prediction endpoint:

- **Endpoint:** http://127.0.0.1:8100/load-model/
- **Request**

```
{
  "text": "What is the capital of india?"
}
```

- **Response:**

```
{
  "output": "New Delhi"
}
```

Error handling

HTTP status codes summary

Status Code	Status Message
200	Status ok
422	Invalid Request
400	Invalid model choice
500	Unable to download the model
500	Unable to load the model
500	Unable to get model prediction.
500	Unexpected error

Models

Support for two large language models has been added in the solution:

1. GPT4All-J : A fine-tuned GPT-J model on assistant style interaction data. The v1.3-groovyGPT-J or GPT-J-6B is an open-source LLM developed. It is a generative pre-trained transformer model designed to produce human-like text that continues from a prompt. The 4-bit quantized model size is 3.7GB.
2. LLama - LLama LLM is a transformer based LLM mode. For our solution the 7 billion parameter pre-trained LLama has been used. The 4-bit quantized model size is 3.9 GB

Challenges

- System memory constraints such as RAM make the inference speed slow.
- While quantization can help reduce the memory footprint and computational requirements of models, it often comes with a trade-off in terms of the model's quality or accuracy.

- Dockerizing LLMs is challenging because of dependency issues on the OS architecture.
- Computational complexity leading to delayed response time from the LLMs.
- Running the dockerized version of the model in M1 Apple silicons can be tricky due to compatibility issues of `ARM` architecture. Using the solution without docker is preferred.
- There is a tradeoff in the performance between using LLMs with lesser parameters vs those with more parameters. Given enough compute resources better and larger LLMs (eg. Falcon, LLama 12B etc) can be used. This would result in improved and sensible responses from the chatbot.

Tools

Tool	Description	Use
GPT4All	Ecosystem of open-source chatbots trained on a massive collection of clean assistant data including code, stories and dialogue.	GPT4All has been used to run the GPT-J LLM.
LangChain	LangChain is a framework for developing applications powered by language models.	Langchain (version 0.0.220) has been used to interact with GPT4all and LLama-cpp. llama-cpp is a Python binding for llama.cpp

FastAPI	It is a modern, fast (as the name implies), web framework for building APIs with Python. It is designed to be easy to use, highly efficient, and scalable.	FastAPI is used to create the backend api(app.py)
Uvicorn	It is an ASGI web server implementation for Python.	It has been used to serve the backend API.
Gradio	It is an open-source Python library that allows building and deploying custom user interfaces (UIs).	The front-end user interface has been built using gradio.

Improvements

- Explore more pre-trained Language Models to identify the most suitable model with better responses.
- Implement additional features in the web application like session storage to enhance user experience and make the chatbot interaction more intuitive.
- Integrate natural language processing techniques to improve the chatbot's understanding and generation of more contextually relevant responses.
- Implement a database or caching mechanism to store frequently used responses and improve response time for repeated queries.
- Conduct load testing to ensure the web application can handle multiple concurrent user interactions efficiently.
- Provide an administrative interface to manage the chatbot's knowledge base, including adding new responses, managing user interactions and flagging inappropriate responses.
- Better and larger LLMs (eg. Falcon, LLama 12B etc) can be used to improve the chatbot performance.