SanjAI

Flask-based Chatbot with Local LLM Integration

TABLE OF CONTENTS

1. PROJECT OVERVIEW

This project focuses on building a **conversational AI chatbot** powered by a locally hosted **Mistral-based language model**. The chatbot is designed to interact with users in real time, process natural language prompts, and generate intelligent responses. Unlike traditional chatbots with fixed responses, this system leverages modern **Large Language Models (LLMs)** to provide context-aware and dynamic replies.

The entire system is divided into three major components:

1. **Backend Layer** – Implemented using Flask (Python) to manage communication between the user interface and the model.
2. **Model Layer** – Based on Hugging Face's transformers library, specifically using AutoModelForCausalLM, which enables efficient inference from the Mistral LLM.
3. **Frontend Layer** – Developed using HTML, CSS, and JavaScript to provide an intuitive and interactive chat interface for end users.

Importance

Conversational AI is becoming a critical component in various fields such as customer service, education, healthcare, and personal assistance. By deploying a locally hosted model, this project also addresses concerns regarding data privacy, security, and internet dependency, since user queries are processed on the system itself rather than through a cloud-based service.

System Components

i. Backend: Flask (Python)

Flask acts as the backbone of the system. It is a lightweight web framework that exposes API endpoints to handle requests and responses.

- **Responsibilities of Flask in this project**:
    - Accept user prompts via a /chat POST endpoint.
    - Pass the prompt to the LLM for processing.
    - Return the generated response in a JSON format.
    - Render the main interface through the / route.

Flask was chosen due to its **simplicity, flexibility, and scalability**, making it suitable for rapid prototyping and integration with machine learning models.

ii. Model: Hugging Face Transformers

The **Mistral-based model** is loaded using Hugging Face's transformers library. Specifically, the AutoModelForCausalLM class is used to load a causal language model capable of predicting the next token in a sequence.

- **Why Mistral?**
    - It is optimized for **efficiency and performance**, suitable for local deployments.
    - It provides **high-quality responses** while requiring fewer resources compared to some other LLMs.
- **Process Flow**:

1. User input is tokenized.
2. The model processes the tokens and predicts the most likely continuation.
3. The output tokens are decoded back into human-readable text.

This enables the chatbot to handle open-domain queries in real time.

iii. Frontend: HTML, CSS, and JavaScript

The frontend serves as the **user interface** where users can interact with the chatbot. It mimics the design of modern chat applications, making it intuitive and user-friendly.

- **Features of the UI**:
    - A chat window displaying user and bot messages.
    - Input box for users to type their queries.
    - Asynchronous communication with the backend using JavaScript (AJAX or Fetch API).
    - Styling with CSS to enhance usability and aesthetics.

This ensures that the chatbot feels **interactive, responsive, and accessible**.

iv. Advantages of the System

- **Privacy-Friendly**: Since the model is locally hosted, user data does not leave the system.
- **Customizable**: Developers can fine-tune the Mistral model or integrate domain-specific datasets.
- **Lightweight and Efficient**: Flask and Mistral provide a good balance between performance and resource consumption.
- **Cross-Platform**: The frontend can be accessed through any web browser.

v. Limitations

- The system requires **sufficient local resources** (RAM and GPU) to handle LLM inference efficiently.
- It may not support extremely large models without hardware optimization.
- Currently limited to **text-based interaction**, lacking voice or multimodal capabilities.

vi. Future Enhancements

Some possible future improvements include:

- Adding **speech-to-text and text-to-speech** modules for voice-based interaction.
- Implementing **conversation memory** for context persistence across multiple queries.
- Optimizing the model for faster inference through quantization or GPU acceleration.
- Extending the chatbot for **domain-specific tasks** such as healthcare guidance, coding help, or educational tutoring.

2. OBJECTIVES

To build a locally hosted AI chatbot that operates without external APIs, offering an interactive real-time chat interface with configurable generation parameters such as tokens, temperature, and top-p. The system is designed with a Flask API integrated into a frontend UI, making it flexible and deployment-ready.

- Build a **local AI chatbot** without relying on external APIs.
- Provide an **interactive chat interface** for real-time conversation.
- Allow configurable **generation parameters** (tokens, temperature, top-p).
- Enable deployment-ready **Flask API integration** with frontend.

3. TECH STACK

- **Languages:** Python, JavaScript, HTML, CSS
- **Frameworks:** Flask
- **Libraries:** Hugging Face Transformers, Torch
- **Frontend:** Vanilla JS, CSS (custom styled chat UI)
- **Model:** Mistral (fine-tuned local LLM)

The chatbot is developed using **Python** (for backend logic, e.g., handling requests), **JavaScript** (for real-time chat interactions), along with **HTML and CSS** (to structure and style the interface). The backend runs on **Flask**, a lightweight framework needed to expose API endpoints and connect the UI with the model. For natural language processing, **Hugging Face Transformers** and **Torch** are used to load and run the **Mistral LLM**, enabling intelligent text generation (e.g., answering questions or summarizing input). On the frontend, **vanilla JavaScript with custom CSS** creates a simple yet interactive chat UI, ensuring smooth user experience without relying on heavy frameworks.

4. SYSTEM ARCHITECTURE

Flow: User (Browser UI) → JavaScript Fetch API → Flask Backend (/chat) → HuggingFace Model (Mistral) → Response → UI Update

- **Frontend:** Renders chat UI, captures user input, sends request to /chat.
- **Flask Backend:** Provides endpoints (/ and /chat) for serving UI and handling requests.
- **Model Integration:** Loads Mistral model from local directory, processes user input, and generates responses.

5. IMPLEMENTATION DETAILS

Backend (Flask API)

- / → Serves index.html chat interface.
- /chat → Accepts POST request with user prompt and generation parameters.
- Calls generate_response() from function.py → Returns AI response as JSON.

Code snippet Flask App- app.py:

```
from flask import Flask, render_template, request, jsonify
from function import generate_response
app = Flask(__name__)
@app.route("/", methods=["GET"])
def home():
    return render_template("index.html")
@app.route("/chat", methods=["POST"])
def chat():
    data = request.get_json(silent=True) or {}
    user_message = (data.get("message") or "").strip()
```

```
    if not user_message:
        return jsonify({"error": "Empty message"}), 400
    try:
        reply = generate_response(
            user_message,
            max_new_tokens=int(data.get("max_new_tokens", 200)),
            temperature=float(data.get("temperature", 0.7)),
            top_p=float(data.get("top_p", 0.9)),
            repetition_penalty=float(data.get("repetition_penalty", 1.05)),
        )
        clean_reply = reply.strip()
        return jsonify({
            "success": True,
            "user_message": user_message,
            "reply": clean_reply
        })
    except Exception as e:
        return jsonify({
            "success": False,
            "error": str(e)
        }), 500
if __name__ == "__main__":
    app.run(host="127.0.0.1", port=5000, debug=True, threaded=True)
```

Model (Hugging Face + Torch)

- Loads **local Mistral model** from MODEL_DIR.
- Dynamically selects device (cuda, mps, or cpu).
- Supports configurable parameters:
    - max_new_tokens
    - temperature
    - top_p
    - repetition_penalty

Code snippet Model Loading:

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
MODEL_DIR = "/Users/yuvaraj/Downloads/sanjai_mistral_model"
if torch.cuda.is_available():
    DEVICE = "cuda"
elif getattr(torch.backends, "mps", None) and torch.backends.mps.is_available():
    DEVICE = "mps"
else:
    DEVICE = "cpu"
tokenizer = AutoTokenizer.from_pretrained(MODEL_DIR, use_fast=True)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token
dtype = torch.float16 if DEVICE == "cuda" else torch.float32
model = AutoModelForCausalLM.from_pretrained(
```

```
    MODEL_DIR,
    torch_dtype=dtype if DEVICE == "cuda" else None,
)
model.to(DEVICE)
model.eval()
@torch.no_grad()
def generate_response(user_input: str,
                      max_new_tokens: int = 200,
                      temperature: float = 0.7,
                      top_p: float = 0.9,
                      repetition_penalty: float = 1.05) -> str:
    prompt = f"<s>[INST] {user_input} [/INST]"
    inputs = tokenizer(prompt, return_tensors="pt", padding=True, truncation=True).to(model.device)
    output_ids = model.generate (**inputs, max_new_tokens=max_new_tokens,
        do_sample=True if temperature > 0 else False,
        temperature=temperature,
        top_p=top_p,
        repetition_penalty=repetition_penalty,
        pad_token_id=tokenizer.pad_token_id or tokenizer.eos_token_id,
        eos_token_id=tokenizer.eos_token_id,
    )
    gen_ids = output_ids[0][inputs["input_ids"].shape[-1]:]
    text = tokenizer.decode(gen_ids, skip_special_tokens=True)
    return text.strip()
```

Frontend (JavaScript)

- Handles user input and sends requests to backend.
- Renders chat bubbles for **user** and **assistant**.
- Displays “Thinking…” while awaiting model response.
- Updates chat window with assistant’s reply.

Code Snippet (Frontend – script.js):

```
const chatBox = document.getElementById("chat-box");
const form = document.getElementById("chat-form");
const input = document.getElementById("user-input");
const sendBtn = document.getElementById("send-btn");
const elMaxNew = document.getElementById("max_new_tokens");
const elTemp   = document.getElementById("temperature");
const elTopP   = document.getElementById("top_p");
function addMessage(role, text, loading=false) {
  const div = document.createElement("div");
  div.className = `msg ${role === "user" ? "user" : "bot"} ${loading ? "loading" : ""}`;
  const roleEl = document.createElement("span");
  roleEl.className = "role";
  roleEl.textContent = role === "user" ? "You" : "Assistant";
  div.appendChild(roleEl);
  div.append(text);
  chatBox.appendChild(div);
  chatBox.scrollTop = chatBox.scrollHeight;
```

```
  return div;
}
async function sendMessage(ev) {
  ev?.preventDefault?.();
  const message = input.value.trim();
  if (!message) return;
  addMessage("user", message);
  input.value = "";
  sendBtn.disabled = true;
  const pending = addMessage("bot", "Thinking…", true);
  try {
    const resp = await fetch("/chat", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        message,
        max_new_tokens: Number(elMaxNew.value || 64),
        temperature: Number(elTemp.value || 0),
        top_p: Number(elTopP.value || 1),
      }),
    });
    const data = await resp.json();
    pending.classList.remove("loading");
    pending.textContent = "";
```

```
    const roleEl = document.createElement("span");
    roleEl.className = "role";
    roleEl.textContent = "Assistant";
    pending.appendChild(roleEl);
    pending.append(data.reply || data.error || "No response");
  } catch (e) {
    pending.classList.remove("loading");
    pending.textContent = "Error: " + (e?.message || e);
  } finally {
    sendBtn.disabled = false;
    input.focus();
  }
}
form.addEventListener("submit", sendMessage);
input.addEventListener("keydown", (e) => {
  if (e.key === "Enter" && (e.ctrlKey || e.metaKey)) {
    sendMessage(e);
  }
});
```

UI/UX (CSS)

- Dark-mode styled chat interface.
- Responsive design with **gradient backgrounds**.
- Separate styling for **user messages** and **bot messages**.
- Interactive send button and input area.

Code Snippet (Styling – style.css)

```
:root {
  --bg: #0f172a;
  --panel: #111827;
  --text: #e5e7eb;
  --muted: #9ca3af;
  --accent: #22d3ee;
  --bubble-user: #1f2937;
  --bubble-bot: #0b5;
}
body {
  background: radial-gradient(1200px 600px at 10% -10%, #0b5 0, transparent 40%),
              radial-gradient(1000px 800px at 110% 0%, #22d3ee22 0, transparent 45%),
              var(--bg);
  color: var(--text);
  font-family: system-ui, sans-serif;
}
.app {
```

```
  max-width: 900px;
  margin: 40px auto;
  background: #0b0f1a88;
  border: 1px solid #243;
  border-radius: 16px;
  overflow: hidden;
}
.chat-box {
  height: 60vh;
  overflow-y: auto;
  padding: 18px;
  background: linear-gradient(180deg, #0a0f1a 0%, #05070c 100%);
}
.msg {
  max-width: 80%;
  padding: 12px 14px;
  border-radius: 14px;
  margin: 10px 0;
  white-space: pre-wrap;
  border: 1px solid #1b2838;}
.user { background: var(--bubble-user); margin-left: auto; }
.bot  { background: #0e1726; border-color: #0a1a28; }
.role { font-size: 12px; opacity: .65; margin-bottom: 4px; display: block; }
.chat-form {
```

```
  display: flex;
  flex-direction: column;
  gap: 8px;
  padding: 12px;
  border-top: 1px solid #112233;
  background: #0a0f1aee;
}
#user-input {
  width: 100%;
  min-height: 54px;
  max-height: 220px;
  padding: 12px 14px;
  border-radius: 12px;
  border: 1px solid #1b2838;
  background: #0e1726;
  color: var(--text);
}
```

6. KEY FEATURES

- Local **LLM integration** (no API dependency)
- Customizable **generation parameters**
- **Interactive real-time chat UI**
- **Error handling** for invalid/empty requests
- **Device adaptive** (CUDA, MPS, CPU)

The chatbot system is designed with several practical features that make it efficient and user-friendly. It integrates a locally hosted LLM, removing the need for external APIs and ensuring data privacy while allowing offline operation. The model supports customizable generation parameters, enabling developers to fine-tune response styles such as creativity, length, or precision. An interactive real-time chat interface enhances the user experience by providing instant, fluid conversations. To maintain reliability, the backend includes error handling mechanisms that manage invalid or empty user requests gracefully. Additionally, the system is device adaptive, supporting CUDA for NVIDIA GPUs, MPS for Apple Silicon, and fallback to CPU, ensuring smooth performance across different hardware environments.

7. CHALLENGES & SOLUTIONS

During development, several challenges were encountered and addressed effectively. One major issue was the slow response time of large model inference, which was solved by optimizing with device-aware data types, using float16 for CUDA-enabled GPUs and float32 for other devices. Another challenge was the absence of a pad token in the tokenizer, which was resolved by setting the pad token equal to the end-of-sequence (EOS) token. Finally, to maintain a responsive UI during model generation, loading indicators and asynchronous JavaScript fetch calls were implemented, ensuring smooth real-time interaction for the user.

- **Challenge:** Large model inference slowed response time
    - Solution: Optimized with device-aware dtype (float16 for CUDA, float32 otherwise).
- **Challenge:** Missing pad token in tokenizer

- Solution: Set pad_token = eos_token.
- **Challenge:** Keeping UI responsive during model generation
- Solution: Added loading indicators and async JS fetch.

8. RESULTS / IMPACT

- Successfully deployed a **local chatbot** with near real-time responses.
- End-to-end integration of **Flask + Hugging Face + JS UI**.
- Flexible design allowing further fine-tuning and deployment on cloud/containers.

The project was successfully implemented as a **local chatbot** capable of generating near real-time responses, showcasing the efficiency of running an advanced LLM without external API dependencies. It achieves **end-to-end integration** by combining Flask for backend communication, Hugging Face Transformers for model inference, and a JavaScript-based UI for an interactive chat experience. The overall design is built to be **flexible and scalable**, supporting further fine-tuning of the model as well as seamless deployment on cloud platforms or within containerized environments, making it adaptable or both personal and enterprise-level applications.

9. FUTURE IMPROVEMENTS

- Add **conversation history** with context retention.
- Implement **authentication & user profiles**.
- Deploy on **Docker + Cloud (AWS/GCP/Azure)**.
- Add support for **multiple models** (switch between GPT-like and Mistral models).
- Optimize performance using **quantization**.

10.REFERENCES & RESOURCES

- Hugging Face Transformers Docs: https://huggingface.co/docs
- Flask Docs: https://flask.palletsprojects.com
- Torch Docs: https://pytorch.org/docs