```
!pip install transformers langchain torch gradio
```

Show hidden output

```
# Install the classic package
!pip install langchain_classic
```

Show hidden output

```
import torch
from transformers import AutoModelForCausalLM,AutoTokenizer
from langchain_classic.memory import ConversationBufferWindowMemory
import gradio as gr
```

```
model_name="meta-llama/Llama-2-7b-chat-hf"
token="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"
```

```
device=torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"using device: {device}")
```

```
using device: cuda
```

```
model=AutoModelForCausalLM.from_pretrained(model_name, token=token).to(device)
tokenizer=AutoTokenizer.from_pretrained(model_name, token=token)
```

```
Loading weights: 100%  291/291 [00:01<00:00, 251.61it/s, Materializing param=model.norm.weight]
generation_config.json: 100%  188/188 [00:00<00:00, 19.3kB/s]
tokenizer_config.json: 100%  1.62k/1.62k [00:00<00:00, 154kB/s]
tokenizer.json: 100%  1.84M/1.84M [00:00<00:00, 2.12MB/s]
tokenizer.model: 100%  500k/500k [00:01<00:00, 424kB/s]
special_tokens_map.json: 100%  414/414 [00:00<00:00, 43.6kB/s]
```

```
def chatbot_responce(user_input):
  # Encode the user's input into tokens, convert to PyTorch tensors, and move to the device (CPU/GPU).
  # max_length and truncation ensure the input fits the model's capacity.
  inputs=tokenizer(user_input,return_tensors="pt",max_length=512,truncation=True).to(device)

  # Generate a response from the loaded pre-trained language model based on the input.
  outputs=model.generate(**inputs)

  # Decode the generated tokens back into a human-readable string, skipping special tokens.
  bot_reply=tokenizer.decode(outputs[0],skip_special_tokens=True)

  # If the bot's reply starts with the user's input (a common artifact in some models),
  # remove the repeated input from the start of the reply.
  if bot_reply.startswith(user_input):
      bot_reply=bot_reply[len(user_input):].strip()
  return bot_reply
```

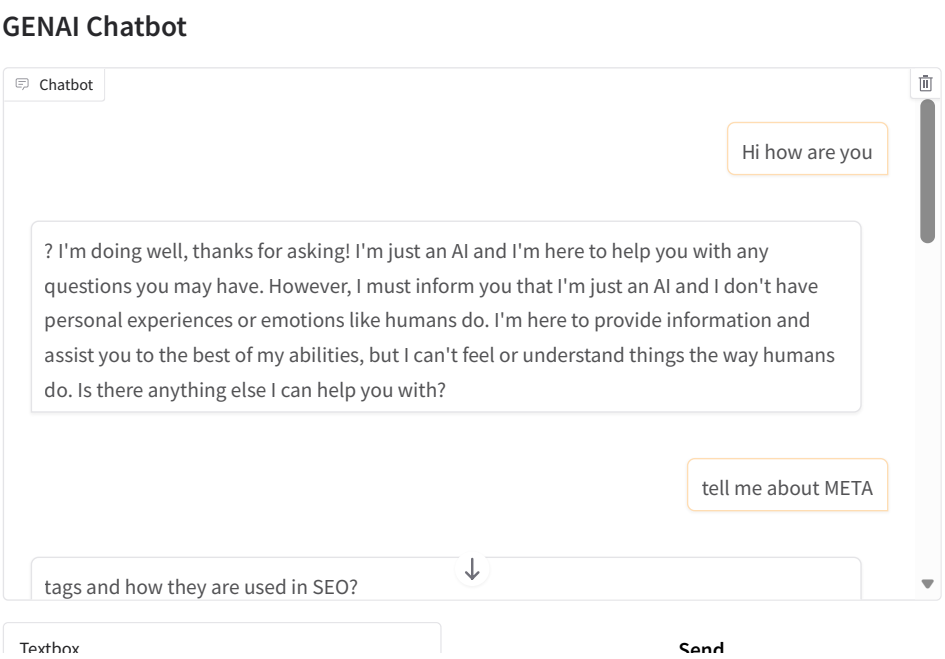
```
def chatbot_interface(user_input,history):
  # Call the core chatbot logic to get a response for the user's input.
  response=chatbot_responce(user_input)
  # Append the user's input and the bot's response to the conversation history.
  history.append((user_input,response))
  # Return the updated history and an empty string to clear the user input box in Gradio.
  return history,""

# Initialize Gradio Blocks for a custom web interface.
demo=gr.Blocks()
```

```
with demo:
  gr.Markdown("## GENAI Chatbot")
  chatbot=gr.Chatbot()
  with gr.Row():
    user_input=gr.Textbox(placeholder="Type your message here",lines=1)
    send_button=gr.Button("Send")

  history=gr.State([])

  send_button.click(chatbot_interface,inputs=[user_input,history],outputs=[chatbot, user_input])
  user_input.submit(chatbot_interface,inputs=[user_input,history],outputs=[chatbot, user_input])
```

```
/tmp/ipython-input-1170503289.py:3: UserWarning: You have not specified a value for the `type` parameter. Defaulting to the
  chatbot=gr.Chatbot()
/tmp/ipython-input-1170503289.py:3: DeprecationWarning: The default value of 'allow_tags' in gr.Chatbot will be changed from
  chatbot=gr.Chatbot()
```

```
demo.launch(share=True)
```

```
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://b47b14568e9f942464.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy` from the terminal in the
```



Next steps: Deploy to Cloud Run

```
from huggingface_hub import login

# Ensure the token variable is available (it should be from previous cells)
if 'token' in locals() or 'token' in globals():
    login(token=token)
    print("Successfully logged in to Hugging Face.")
else:
    print("Hugging Face token not found. Please ensure it's defined.")
```

```
Successfully logged in to Hugging Face.
```

```
from huggingface_hub import HfApi

try:
    api = HfApi(token=token)
    user_info = api.whoami()
    print(f"Successfully authenticated as: {user_info['name']}")
    # Further check for model access if needed, e.g., by trying to fetch model info
    model_info = api.model_info(model_name)
    print(f"Successfully retrieved info for model: {model_info.modelId}")
    print("Your Hugging Face token is valid and has access to the specified model.")
except Exception as e:
    print(f"Error during Hugging Face token verification or model access check: {e}")
    print("Please ensure your token is correct, has the necessary permissions, and you have accepted the model's terms on H
```

```
Successfully authenticated as: SurajRavirala
Successfully retrieved info for model: meta-llama/Llama-2-7b-chat-hf
Your Hugging Face token is valid and has access to the specified model.
```