<AI club>
<Build your own mini Jarvis>

Welcome to `AI club's` First 2026 workshop

Let's get started! ⟶

Outline for `Today`

{01} Naviage Google colab

Create a google colab account and start coding!

{02} Build your own Jarvis

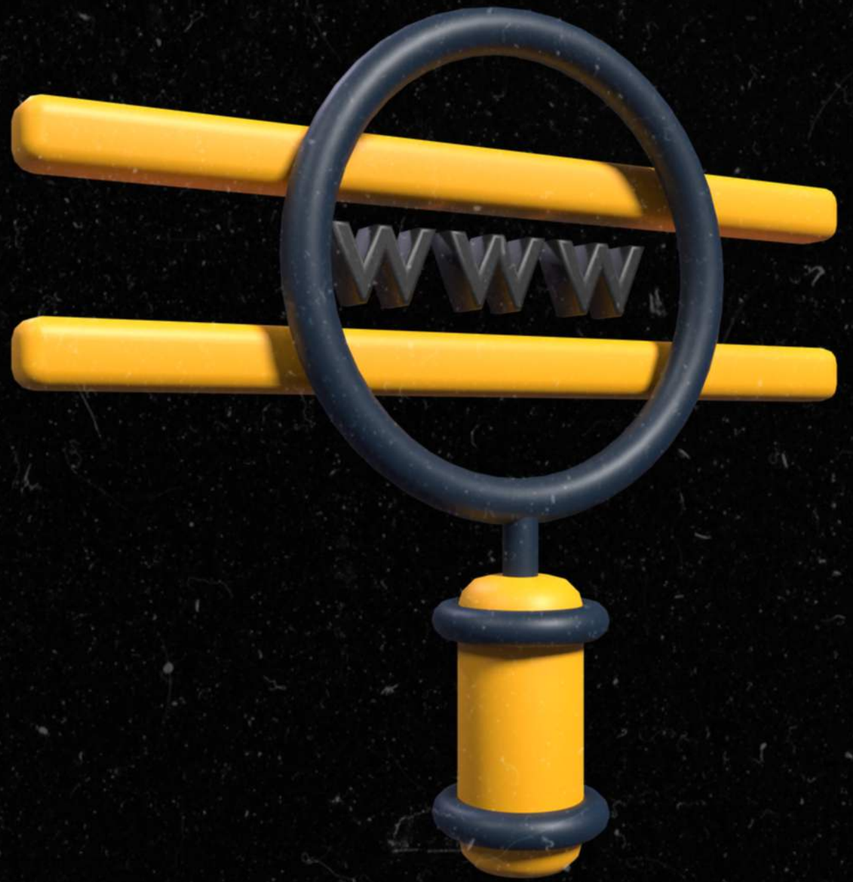
Build your own Javis using a pre-existing AI chatbot

{03} Upload to github

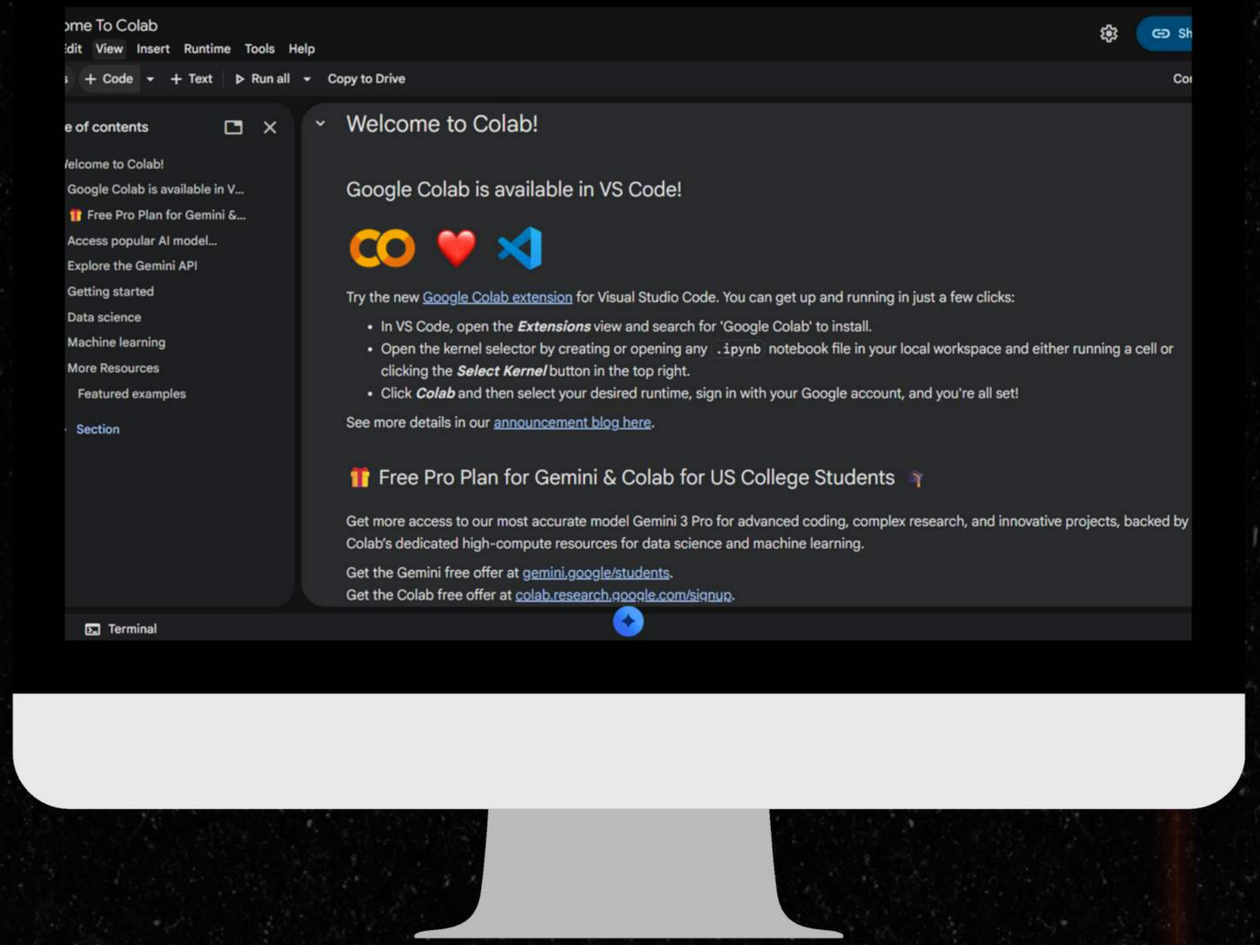
Create a Github repo and upload the project

But First, What is `Google colab`

- Google Colab, short for Colaboratory, is a free cloud-based platform that allows you to write and execute Python code directly in your browser without any setup required. Built on Jupyter notebooks, it provides free access to computing resources including GPUs and TPUs, making it particularly popular for machine learning, data analysis, and educational purposes.

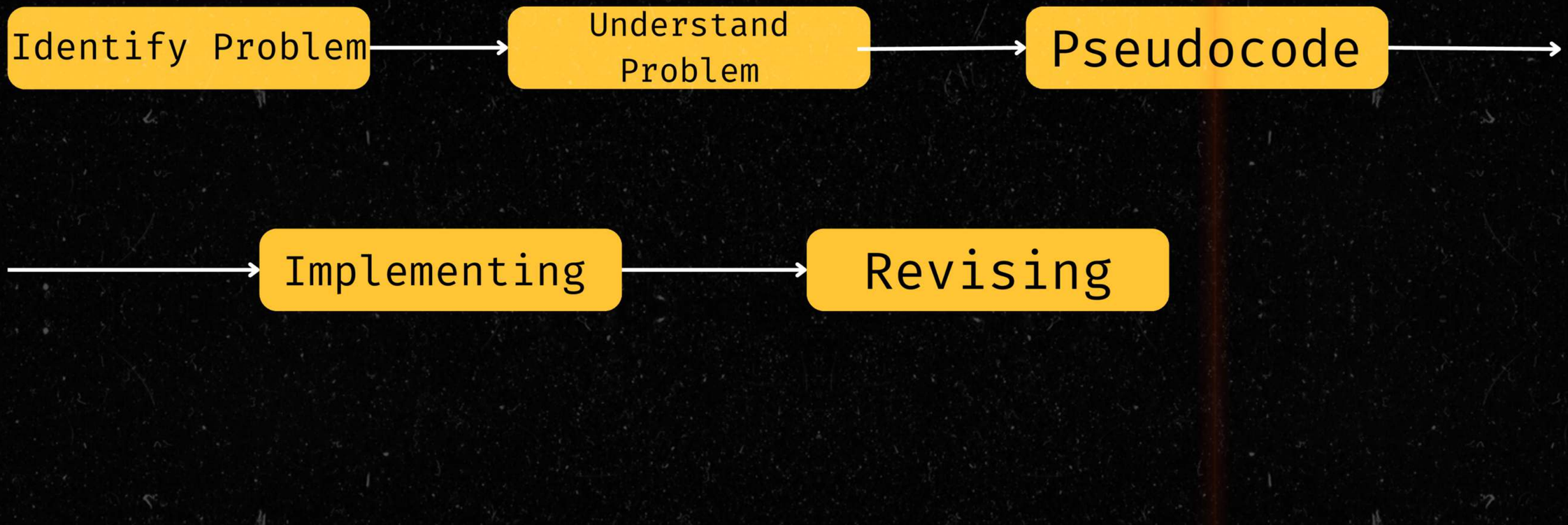


<AI Club> <Build your own mini Jarvis>



<Google colab Homepage>

Structure of Problem Solving



Identify Problem

- What do you want your assistant to do?
- Answer questions?
- Control smart home devices?
- Manage calendar/reminders?
- Search the web?
- Have conversations?
-

Understand Problem

- Input: Text or voice from user
- Processing: AI model understands and generates response
- Output: Text response + optional voice output
- Memory: Remember conversation context
- Personality: Should feel like talking to JARVIS, not a robot

Pseudocode

```
START
  Load AI model

  WHILE user wants to chat:
    Get user input (text or voice)

    IF has conversation history:
      Add history to context

    Send input + context to AI model
    Get AI response

    Convert response to speech (optional)
    Display response to user

    Save to conversation history
  END WHILE
END
```

Implementing

- Choose tools: Transformers (AI), Gradio (UI), gTTS (voice)
- Pick model: Flan-T5 or Blenderbot
- Build interface: Chat window, input box, buttons
- Add features: Voice output, memory, examples
- Test it: Try different questions

Revising

Test and improve:

- Is the model smart enough? → Try bigger model
- Responses too robotic? → Adjust personality prompt
- Too slow? → Use smaller/faster model
- Need voice input? → Add speech-to-text
- Want it to do actions? → Add function calling

Step 1: Importing

```
!pip install transformers gradio torch accelerate gTTS -q

# STEP 2: Import libraries
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, pipeline
import gradio as gr
import torch
from gtts import gTTS
import os
```

```
transformers - Hugging Face library with pre-trained AI models
gradio - Creates web interfaces for your AI (the chat window)
torch - PyTorch, the deep learning framework that runs the models
accelerate - Makes models load faster and use less memory
gTTS - Google Text-to-Speech, converts text to voice
-q - "quiet mode" so it doesn't spam you with installation details
```

Step 1 :explanation

```
transformers - Hugging Face library with pre-trained AI models

gradio - Creates web interfaces for your AI (the chat window)

torch - PyTorch, the deep learning framework that runs the models

accelerate - Makes models load faster and use less memory

gTTS - Google Text-to-Speech, converts text to voice

-q - "quiet mode" so it doesn't spam you with installation details
```

```
gradio - Builds the chat interface with buttons, text boxes, and voice player

torch - The engine that runs the AI model calculations (like TensorFlow but PyTorch)

gTTS - The Google Text-to-Speech class that turns your assistant's text responses into audio files

os - Operating system tools (used for file handling, like saving audio files)
```

```
AutoTokenizer - Converts text into numbers the AI can understand (and back)

AutoModelForSeq2SeqLM - Loads sequence-to-sequence models (chat models that take text input and
generate text output)

pipeline - Quick shortcut for using models (we might not use this one)
```

Step 2: Loading the AI Model

```
print("Loading your AI assistant... (this might take a minute)")


model_name = "google/flan-t5-large"  # Smarter model!
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

print("Assistant loaded! Ready to chat.")

# STEP 4: Create the chat function with better prompting
def chat_with_assistant(user_input, history):
    # Build conversation context with JARVIS personality
    context = "You are JARVIS, an intelligent and helpful AI assistant. Be conversational and friendly.\n\n"

    # Add conversation history
    if history:
        for human, assistant in history[-3:]:  # Keep last 3 exchanges
            context += f"Human: {human}\nJARVIS: {assistant}\n"

    # Add current question
    context += f"Human: {user_input}\nJARVIS:"
```

Step 2: Explanation

```
flan-t5-large = Google's smart conversational model (780 million parameters)

tokenizer =  AutoTokenizer.from_pretrained(model_name)

Downloads the tokenizer for this specific model
Tokenizer's job: Convert words → numbers (and numbers → words)
Example: "Hello" → [8774, 0] (the model only understands numbers)

model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

Downloads the actual AI brain (the neural network)
Seq2SeqLM = Sequence-to-Sequence Language Model (takes text in, gives text out)
This is the heavy part - downloads ~3GB of model weight

Creating the Chat Function
pythondef chat_with_assistant(user_input, history):

Function that takes user's message and conversation history
Returns the AI's response

context = "You are JARVIS, an intelligent and helpful AI assistant. Be conversational and
friendly.\n\n"

System prompt that tells the AI how to behave
This gives it personality! Without this, it might be bland or robotic
The \n\n adds blank lines for formatting
```

Step 2: Explanation

```
if history:
    for human, assistant in history[-3:]:

Checks if there's previous conversation
[-3:] = only keep the last 3 exchanges (to avoid overwhelming the model)
Example: If you've chatted 10 times, it only remembers the last 3

pythoncontext += f"Human: {human}\nJARVIS: {assistant}\n"

Adds each past exchange to the context
Format: "Human: [their message]\nJARVIS: [AI response]"
This helps the AI remember what you talked about

pythoncontext += f"Human: {user_input}\nJARVIS:"

Adds the current question at the end
Ends with "JARVIS:" so the AI knows it should respond as JARVIS
```

Step 2: Explanation

```
if history:
    for human, assistant in history[-3:]:

Checks if there's previous conversation
[-3:] = only keep the last 3 exchanges (to avoid overwhelming the model)
Example: If you've chatted 10 times, it only remembers the last 3

pythoncontext += f"Human: {human}\nJARVIS: {assistant}\n"

Adds each past exchange to the context
Format: "Human: [their message]\nJARVIS: [AI response]"
This helps the AI remember what you talked about

pythoncontext += f"Human: {user_input}\nJARVIS:"

Adds the current question at the end
Ends with "JARVIS:" so the AI knows it should respond as JARVIS
```

Step 3: Tokenize

```
# Tokenize and generate with better parameters
    inputs = tokenizer(context, return_tensors="pt", truncation=True, max_length=1024)
    outputs = model.generate(
        **inputs,
        max_length=256,  # Longer responses
        num_beams=4,     # Better quality
        temperature=0.7,  # More creative
        do_sample=True,
        top_p=0.9
    )
    response = tokenizer.decode(outputs[0], skip_special_tokens=True)

    return response
```

```
Example: "Hello JARVIS" → tensor([[8774, 0, 446, 4211, 5711]])

pythonoutputs = model.generate(**inputs, ...)
This is where the magic happens - the AI generates a response!
Parameters explained:
pythonmax_length=256

AI can generate up to 256 tokens (~200 words)
Longer = more detailed answers, but takes more time

pythonnum_beams=4

Beam search - explores 4 different possible responses simultaneously
Picks the best one
Higher = better quality but slower (1 = fastest, 5+ = high quality)

pythontemperature=0.7

Controls randomness/creativity
0.1 = very focused, predictable, boring
0.7 = balanced (good default)
1.5 = very creative, sometimes weird
Think of it like "how wild should the AI be?"

pythondo_sample=True

Enables random sampling (needed for temperature to work)
False = always picks the most likely word (boring)
True = sometimes picks less likely words (more natural)

pythontop_p=0.9

Nucleus sampling - only considers words that make up the top 90% probability
Prevents the AI from saying really weird/unlikely things
0.9 = good balance between creativity and coherence


pythonresponse = tokenizer.decode(outputs[0], skip_special_tokens=True)

Converts the AI's output numbers back into readable text
skip_special_tokens=True - removes weird tokens like <pad>, </s>
Example: [1782, 0, 27, 31, 3, 9, ...] → "Hi! I'm an AI assistant..."

pythonreturn response

Sends the text response back
```

Step 3: explanation

```
Enables random sampling (needed for temperature to work)
False = always picks the most likely word (boring)
True = sometimes picks less likely words (more natural)

pythontop_p=0.9

Nucleus sampling - only considers words that make up the top 90% probability
Prevents the AI from saying really weird/unlikely things
0.9 = good balance between creativity and coherence


pythonresponse = tokenizer.decode(outputs[0], skip_special_tokens=True)

Converts the AI's output numbers back into readable text
skip_special_tokens=True - removes weird tokens like <pad>, </s>
Example: [1782, 0, 27, 31, 3, 9, ...] → "Hi! I'm an AI assistant..."

pythonreturn response

Sends the text response back
```

Step 3: explanation

```python
# STEP 5: Create Gradio interface with voice
def gradio_chat_with_voice(message, history):
    response = chat_with_assistant(message, history)

    # Convert to speech
    try:
        tts = gTTS(text=response, lang='en', slow=False)
        audio_file = "response.mp3"
        tts.save(audio_file)
        return response, audio_file
    except Exception as e:
        print(f"TTS Error: {e}")
        return response, None
```

Step 4: Creating interface with voice

```
def gradio_chat_with_voice(message, history):
    response = chat_with_assistant(message, history)

Calls the chat function to get the AI's text response


pythontry:
    tts = gTTS(text=response, lang='en', slow=False)

Creates a Google Text-to-Speech object
text=response - the AI's answer
lang='en' - English language
slow=False - normal speed (True would be slower, clearer)

pythonaudio_file = "response.mp3"
tts.save(audio_file)

Saves the speech as an MP3 file called "response.mp3"
This file gets overwritten each time (so it doesn't fill up your storage)

pythonreturn response, audio_file

Returns both the text AND the audio file
Gradio will display the text in chat and play the audio

pythonexcept Exception as e:
    print(f"TTS Error: {e}")
    return response, None

If voice conversion fails (network issues, etc.), still return the text
None means "no audio file" - so chat still works without voice
```

Step 4: Explication

```
with gr.Blocks(theme=gr.themes.Soft()) as demo:
    gr.Markdown("""
    # 🤖 Your Personal JARVIS Assistant
    ### Built in Google Colab with AI superpowers!
    Ask me anything - I'll respond with text and voice.
    """)

    with gr.Row():
        with gr.Column(scale=2):
            chatbot = gr.Chatbot(
                height=500,
                bubble_full_width=False
            )

            with gr.Row():
                msg = gr.Textbox(
                    label="Your message",
                    placeholder="Ask me anything...",
                    scale=4
                )
                submit = gr.Button("Send 🚀", scale=1, variant="primary")

            clear = gr.Button("Clear Chat 🗑️")

        with gr.Column(scale=1):
            audio_output = gr.Audio(
                label="🔊 Voice Response",
                autoplay=True
            )

            gr.Markdown("### Quick Examples:")
            example_btns = [
                gr.Button("👋 Introduce yourself"),
                gr.Button("😂 Tell me a joke"),
                gr.Button("🧠 Explain AI simply"),
                gr.Button("💡 Give me a fun fact"),
            ]
```

Step 5: Explication

Step 5: Explication

```
with gr.Blocks(theme=gr.themes.Soft()) as demo:

gr.Blocks() - Creates a custom layout (vs simple chat interface)
theme=gr.themes.Soft() - Uses a soft, modern color scheme
as demo - Names this interface "demo" so we can launch it later

gr.Markdown() - Displays formatted text (supports emojis, headers, etc.)

with gr.Row():

Creates a horizontal row - splits screen left/right
Everything inside will be side-by-side

pythonwith gr.Column(scale=2):

Creates a vertical column on the left
scale=2 - takes up 2/3 of the width (bigger than the right column)


Chat Components (Left Side)
pythonchatbot = gr.Chatbot(
    height=500,
    bubble_full_width=False
)

gr.Chatbot() - The actual chat window showing conversation
height=500 - 500 pixels tall
bubble_full_width=False - Chat bubbles don't stretch full width (looks better)

pythonwith gr.Row():
    msg = gr.Textbox(
        label="Your message",
        placeholder="Ask me anything...",
        scale=4
    )
    submit = gr.Button("Send 🚀", scale=1, variant="primary")

Another row (input box + button side by side)
msg - Text input where user types
placeholder - Gray hint text before user types
scale=4 - textbox takes 4/5 of this row
submit - Send button takes 1/5
variant="primary" - Makes button blue/highlighted
```

```
clear = gr.Button("Clear Chat 🗑️")

Button to reset the conversation


Voice & Examples (Right Side)
pythonwith gr.Column(scale=1):

Right column, takes up 1/3 of width (smaller than left)

pythonaudio_output = gr.Audio(
    label="🔊 Voice Response",
    autoplay=True
)

gr.Audio() - Audio player component
autoplay=True - Automatically plays when AI responds (no manual click needed)
```

Step 5: Explication

```
def respond(message, chat_history):
        if not message.strip():
            return "", chat_history, None

        bot_response, audio_file = gradio_chat_with_voice(message, chat_history)
        chat_history.append((message, bot_response))
        return "", chat_history, audio_file

    # Wire up events
    msg.submit(respond, [msg, chatbot], [msg, chatbot, audio_output])
    submit.click(respond, [msg, chatbot], [msg, chatbot, audio_output])
    clear.click(lambda: [], None, chatbot, queue=False)

    # Example button handlers
    example_btns[0].click(lambda: "Introduce yourself", None, msg)
    example_btns[1].click(lambda: "Tell me a joke", None, msg)
    example_btns[2].click(lambda: "Explain artificial intelligence in simple terms", None, msg)
    example_btns[3].click(lambda: "Give me an interesting fun fact", None, msg)

# Launch it!
print("\n🚀 Launching your AI assistant...")
demo.launch(share=True, debug=True)
```

Step 6: Explication

```
def respond(message, chat_history):

Takes the user's message and the full chat history
This is the main function that handles each interaction

pythonif not message.strip():
    return "", chat_history, None

message.strip() - removes spaces from beginning/end
If message is empty (user just hit enter), do nothing
Returns: empty textbox, unchanged history, no audio

pythonbot_response, audio_file = gradio_chat_with_voice(message, chat_history)

Calls your AI function to get the response
Returns both text response AND audio file

pythonchat_history.append((message, bot_response))

Adds this exchange to the conversation history
Format: (user_message, ai_response) as a tuple
Example: [("Hi", "Hello!"), ("How are you?", "I'm great!")]

pythonreturn "", chat_history, audio_file

"" - Clears the input textbox (ready for next message)
chat_history - Updated history (shows in chat window)
audio_file - The MP3 file to play
```

Step 6: Explication

```
msg.submit - Triggered when user hits Enter in textbox
respond - Function to call
[msg, chatbot] - INPUTS: Get text from msg, get history from chatbot
[msg, chatbot, audio_output] - OUTPUTS: Update textbox, chat window, and audio player

pythonsubmit.click(respond, [msg, chatbot], [msg, chatbot, audio_output])
When user clicks "Send 🚀" button:

Same as above, but triggered by button click instead of Enter
Now users can either press Enter OR click the button!

pythonclear.click(lambda: [], None, chatbot, queue=False)
When user clicks "Clear Chat 🗑️":

lambda: [] - Returns empty list (clears history)
None - No inputs needed
chatbot - Updates the chat window to be empty
queue=False - Runs immediately without waiting


Example Button Handlers
pythonexample_btns[0].click(lambda: "Introduce yourself", None, msg)
When user clicks "👋 Introduce yourself":

lambda: "Introduce yourself" - Returns this text
None - No inputs needed
msg - Puts the text into the textbox
User still needs to click Send (it just fills the textbox for them)

pythonexample_btns[1].click(lambda: "Tell me a joke", None, msg)
example_btns[2].click(lambda: "Explain artificial intelligence in simple terms", None, msg)
example_btns[3].click(lambda: "Give me an interesting fun fact", None, msg)

Same pattern for the other 3 buttons
Each puts a different pre-written prompt into the textbox
```

Step 6: Explication

```
msg.submit - Triggered when user hits Enter in textbox
respond - Function to call
[msg, chatbot] - INPUTS: Get text from msg, get history from chatbot
[msg, chatbot, audio_output] - OUTPUTS: Update textbox, chat window, and audio player

pythonsubmit.click(respond, [msg, chatbot], [msg, chatbot, audio_output])
When user clicks "Send 🚀" button:

Same as above, but triggered by button click instead of Enter
Now users can either press Enter OR click the button!

pythonclear.click(lambda: [], None, chatbot, queue=False)
When user clicks "Clear Chat 🗑️":

lambda: [] - Returns empty list (clears history)
None - No inputs needed
chatbot - Updates the chat window to be empty
queue=False - Runs immediately without waiting


Example Button Handlers
pythonexample_btns[0].click(lambda: "Introduce yourself", None, msg)
When user clicks "👋 Introduce yourself":

lambda: "Introduce yourself" - Returns this text
None - No inputs needed
msg - Puts the text into the textbox
User still needs to click Send (it just fills the textbox for them)

pythonexample_btns[1].click(lambda: "Tell me a joke", None, msg)
example_btns[2].click(lambda: "Explain artificial intelligence in simple terms", None, msg)
example_btns[3].click(lambda: "Give me an interesting fun fact", None, msg)

Same pattern for the other 3 buttons
Each puts a different pre-written prompt into the textbox
```

Step 6: Explication

```
print("\n🚀 Launching your AI assistant...")
demo.launch(share=True, debug=True)
```
- `demo.launch()` - Starts the web server
- `share=True` - Creates a **public URL** (like `https://abc123.gradio.live`) that anyone can access for 72 hours
- `debug=True` - Shows detailed error messages if something breaks

Event Flow Diagram
```
User types "Hello" and presses Enter
        ↓
msg.submit triggered
        ↓
respond() called with ("Hello", [])
        ↓
gradio_chat_with_voice() generates response
        ↓
Returns ("Hi there!", "response.mp3")
        ↓
Chat history updated: [("Hello", "Hi there!")]
        ↓
UI updates:
  - Textbox clears ("")
  - Chat shows conversation
  - Audio plays "Hi there!"
```