

# Build an AI Meeting Assistant: Instant Notes, Zero Worries

**Estimated Effort:** 45 minutes

In today's fast-paced work environments, creativity and collaboration drive innovation. Teams often participate in brainstorming sessions, strategy meetings, and problem-solving discussions, which are essential for shaping projects and organizational goals. However, the speed and volume of these interactions can pose significant challenges. Key ideas, critical decisions, and actionable tasks often slip through the cracks when relying on traditional manual note-taking methods. With so much information being exchanged, capturing every detail accurately becomes difficult, leading to unclear follow-ups, missed opportunities, and inefficiencies.

To address these challenges, the **AI-Powered Meeting Assistant** project offers a modern, transformative solution for meeting documentation. By leveraging large language models (LLMs) and generative AI, this project equips participants with the tools to automate note-taking and extract critical insights efficiently.

Using advanced natural language processing, generative AI identifies and captures key meeting elements in real-time, including ideas, decisions, action items, and deadlines. This eliminates the need for manual note-taking, allowing team members to focus fully on the conversation without worrying about missing important details. The AI not only transcribes discussions but also organizes the content into clear, structured, and actionable meeting minutes. These organized records serve as a reliable resource that enhances clarity, improves follow-ups, and ensures teams stay aligned and productive as they move forward confidently.

In this project, you will:

- **Familiarize yourself with Whisper:** Begin by exploring Whisper, an application that converts audio speech to text. Understand its features, functionality, and limitations to ensure accurate transcription capabilities.
- **Build an initial app using Gradio:** Create a simple app with Gradio, a Python library for building web-based interfaces. This step introduces you to handling audio inputs and outputs, preparing you for more advanced functionality.
- **Develop a complete speech-to-text application:** Leverage your knowledge of Whisper and Gradio to design a robust speech-to-text application. Ensure it supports multiple audio formats and is user-friendly.
- **Explore IBM watsonx.ai LLM:** Familiarize yourself with IBM watsonx.ai large language models (LLMs) to add advanced natural language processing features. Use these capabilities to improve the app's handling of transcriptions, such as context-aware corrections.
- **Preprocess speech-to-text transcripts:** Clean and preprocess the transcriptions generated by the app. This includes adding proper punctuation, correcting errors, and structuring the text for readability and further analysis.
- **Utilize PromptTemplate and Chain:** Incorporate PromptTemplate and Chain (from LangChain) to create advanced workflows. Use these tools to extend the app's functionality, such as generating summaries, extracting key points, or translating transcriptions.
- **Integrate and finalize:** Bring all components together—Whisper, Gradio, IBM watsonx.ai LLM, and PromptTemplate workflows—into a cohesive Gradio application. The final product should deliver an end-to-end solution for converting and processing audio into polished, usable text.

Note: This guided project is an updated version of this project with newer models and added features: [AI meeting companion: From voice to insight](#)

## Tips for the best experience! - Read Only

Keep the following tips handy and refer to them at any point of confusion throughout the lab. Do not worry if they seem irrelevant now. You will go through everything step by step later.

Please note the content on this page is for read-only. Do NOT try to run any code here.

- At any point throughout the lab, if you are lost, click on **Table of Contents** icon on the top left of the page and navigate to your desired content.

- Whenever you make changes to a file, be sure to save your work. Cloud IDE automatically saves any changes you make to your files immediately. You can also save from the toolbar.
- After setting up your virtual environment in the next step, be sure to always have it activated for the rest of the lab. You'll know it's activated when you see (my\_env) before the prompt, like this:
- If (my\_env) doesn't appear, activate the environment by running the following command in the terminal:

```
source my_env/bin/activate
```

- For running the application, always ensure application file is running in the background before opening the Web Application.
- You run a code block by clicking >\_ on bottom right.

```
python app.py
```

- Always ensure that your current directory is /home/project. If you are not in the correct folder, certain code files may fail to run. Use the cd command to navigate to the correct location.
- Make sure you are accessing the application through port 5000. Clicking the purple Web Application button will run the app through port 5000 automatically.

#### **Web Application**

**Note:** If this "Web Application" button does not work, follow the following picture instructions to launch the application.

- If you get an error about not being able to access another port (for example, 8888), just refresh the app by clicking the small refresh icon. In case of other errors related to the server, simply refresh the page as well. (The following image is for illustration purpose only.)
- To stop execution of app.py in addition to closing the application tab, press Ctrl+C in terminal.
- If you encounter an error running the application or after you entered your desired keyword, try refreshing the app using the button on the top of the application's page. You can try inputting a different query too.
- Typically, using the models provided by watsonx.ai would require watsonx credentials, including an API key and a project ID. However, in this lab, these credentials are not needed.

## Setting up your development environment

### Preparing the environment

1. Start setting up the environment by creating a Python virtual environment and installing the required libraries. Use the following commands in the terminal:

```
pip3 install virtualenv
virtualenv my_env # create a virtual environment my_env
source my_env/bin/activate # activate my_env
```

2. Install the required libraries in the environment (NOTE: This will take several minutes. Please be patient. You can go grab a quick coffee and come back!).

```
# installing required libraries in my_env
pip install transformers==4.35.2 \
torch==2.1.1 \
```

```
gradio==5.9.0 \
langchain==0.3.12 \
langchain-community==0.3.12 \
langchain_ibm==0.3.5 \
ibm-watsonx-ai==1.1.16 \
pydantic==2.10.3
```

3. Install `ffmpeg` so that you can work with audio files in Python. Run this command first to update the package lists for the Advanced Package Tool.

```
sudo apt update
```

4. Then, install `ffmpeg`.

```
sudo apt install ffmpeg -y
```

## Step 1. Speech-to-text

Initially, you want to create a simple speech-to-text Python file by using OpenAI Whisper. We provide a [sample audio file](#) for testing.

1. Create and open a Python file and call it `simple_speech2text.py` by clicking the following button:

[Open `simple\_speech2text.py` in IDE](#)

2. Copy the following code into `simple_speech2text.py`.

```
import requests
# URL of the audio file to be downloaded
url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/hTqGqoC-LrW6S79HjuJUkg(trimmed-02.wav"
# Send a GET request to the URL to download the file
response = requests.get(url)
# Define the local file path where the audio file will be saved
audio_file_path = "sample-meeting.wav"
# Check if the request was successful (status code 200)
if response.status_code == 200:
    # If successful, write the content to the specified local file path
    with open(audio_file_path, "wb") as file:
        file.write(response.content)
        print("File downloaded successfully")
else:
    # If the request failed, print an error message
    print("Failed to download the file")
```

3. Run the Python file to download the sample audio file and place it in our working directory.

```
python3 simple_speech2text.py
```

You should see the downloaded audio file in the file explorer window.

4. Implement [OpenAI Whisper](#) to transcribe voice to speech. **You can override the previous code in the Python file.**

```
import torch
from transformers import pipeline
# Initialize the speech-to-text pipeline from Hugging Face Transformers
# This uses the "openai/whisper-tiny.en" model for automatic speech recognition (ASR)
# The `chunk_length_s` parameter specifies the chunk length in seconds for processing
pipe = pipeline(
    "automatic-speech-recognition",
    model="openai/whisper-tiny.en",
    chunk_length_s=30,
)
# Define the path to the audio file that needs to be transcribed
sample = 'sample-meeting.wav'
# Perform speech recognition on the audio file
# The `batch_size=8` parameter indicates how many chunks are processed at a time
# The result is stored in `prediction` with the key "text" containing the transcribed text
prediction = pipe(sample, batch_size=8)["text"]
# Print the transcribed text to the console
print(prediction)
```

5. Run the Python file, and you should get the output below in the terminal.

```
python3 simple_speech2text.py
```

Wasn't that easy? In the next step, you use Gradio to create the interface for your app.

## Gradio interface

### Create a simple demo

Throughout this project, you will create different large language model (LLM) applications with a Gradio interface. So, let's first get familiar with Gradio by creating a simple app.

1. Still in the project directory, create a Python file and name it `hello.py`. You can click on the button below to do so.

[Open hello.py in IDE](#)

2. Open the `hello.py` file, paste the following Python code, and save the file.

```
import gradio as gr
def greet(name):
    return "Hello " + name + "!"
demo = gr.Interface(fn=greet, inputs="text", outputs="text")
demo.launch(server_name="0.0.0.0", server_port= 5000)
```

The previous code creates a `gradio.Interface` called `demo`. It wraps the `greet` function with a simple text-to-text user interface that you could interact with.

The `gradio.Interface` class is initialized with three required parameters:

- fn: The function to wrap a UI around
- inputs: Which components to use for the input (for example, “text”, “image” or “audio”)
- outputs: Which components to use for the output (for example, “text”, “image” or “label”)

The last line, `demo.launch()`, launches a server to serve your `demo`.

## Launching the demo app

Now, go back to the terminal and ensure that the `my_env` virtual environment name is displayed at the beginning of the line. Then, run the following command to execute the Python script.

```
python3 hello.py
```

As the Python code is served by the local host, click the following button, and you can see the simple application that you just created. Feel free to play around with the input and output of the web app!

Click the following button to see the application.

**Web Application**

**Note:** If this “Web Application” button does not work, follow the following picture instructions to launch the application.

You should see the following image. In this example, bob is the name entered.

When you're done and want to exit, **press Ctrl+C** in the terminal and close the application tab.

You just got a small example of the Gradio interface. It's easy, right? If you want to learn a little bit more about customization in Gradio, you can checkout their [Quickstart guide](#).

For the rest of this project, you use Gradio as an interface for LLM apps.

## Step 2: Creating audio transcription app

Now, let's create a new Python file `speech2text_app.py` by clicking the following button:

**Open speech2text\_app.py in IDE**

In this file, we will implement a function called `transcript_audio` which we went over in step 1.

**Excercise: Complete the `transcript_audio` function below. After completing it, copy paste it in your `speech2text_app.py` file and save the changes.**

1. From Step 1, fill in the missing parts in the `transcript_audio` function below.

```
import torch
from transformers import pipeline
import gradio as gr
# Function to transcribe audio using the OpenAI Whisper model
def transcript_audio(audio_file):
    # Initialize the speech recognition pipeline
    pipe = #-----> Fill here <-----
    # Transcribe the audio file and return the result
    result = #-----> Fill here <-----
    return result
# Set up Gradio interface
audio_input = gr.Audio(sources="upload", type="filepath") # Audio input
output_text = gr.Textbox() # Text output
# Create the Gradio interface with the function, inputs, and outputs
iface = gr.Interface(fn=transcript_audio,
                      inputs=audio_input, outputs=output_text,
                      title="Audio Transcription App",
                      description="Upload the audio file")
# Launch the Gradio app
iface.launch(server_name="0.0.0.0", server_port=5000)
```

You can find the solution below.

► Click for the solution

2. Save your file and run the script by clicking the button below.

```
python3 speech2text_app.py
```

3. Start the app by clicking the following button.

**Web Application**

You can download the sample audio file that's provided by right-clicking on it in the file explorer and selecting **Download**. After it's downloaded, you can upload this file to the app.

Alternatively, you can choose and upload any MP3/WAV audio file from your system.

Upload your audio file then select **Submit** to get the transcript.

**Note:** Occasionally, the CloudIDE may go offline after a file upload. If this happens, wait for 2-3 minutes for the IDE to come back online, then click **Submit**. If it takes longer, try refreshing your browser.

The result will be:

4. Press **Ctrl+C** to stop the application.

## Step 3. Integrating LLM using IBM watsonx Granite

### Running a simple LLM

Ready to generate text with LLMs? You'll start by generating text with LLMs.

Create a Python file and name it `simple_llm.py`. You can continue by clicking the following button or by referencing the accompanying image.

**Open simple\_llm.py in IDE**

In this task, we use `ibm/granite-3-3-8b-instruct` model as an LLM instance. You can try out other models available on [IBM watsonx.ai](#).

#### How the code works:

##### 1. Importing required libraries:

The code begins by importing libraries and modules required for interacting with IBM watsonx foundation models and LangChain:

- `ModelTypes`, `DecodingMethods`, `EmbeddingTypes`: For defining model types, decoding methods, and embedding types.
- `APIClient` and `Credentials`: For managing API interactions and credentials.
- `GenTextParamsMetaNames`: For managing model parameters such as decoding methods, maximum tokens, and temperature.
- `WatsonxLLM` and `WatsonxEmbeddings`: For interacting with IBM's large language models and embedding features.
- `PromptTemplate` and `LLMChain`: For defining prompt templates and chaining operations.

##### 2. Defining model parameters:

The `parameters` dictionary is configured to customize the behavior of the language model:

- `DECODING_METHOD`: "sample": Specifies the sampling that will be used for generating responses.
- `MAX_NEW_TOKENS`: 512: Limits the maximum number of tokens the model can generate in one response.
- `MIN_NEW_TOKENS`: 1: Sets the minimum number of tokens for a response.
- `TEMPERATURE`: 0.5: Controls the randomness of the model's output; lower values make responses more deterministic.
- `TOP_K`: 50 and `TOP_P`: 1: Define additional parameters for sampling strategies.

##### 3. Setting model and project IDs:

- `model_id`: Specifies the model being used, in this case, `ibm/granite-3-3-8b-instruct`, an 8-billion parameter model designed for instructional tasks.
- `project_id`: Defines the project under which the interaction is taking place, e.g., "skills-network".

##### 4. Initializing the WatsonxLLM model:

- A `WatsonxLLM` object, `granite_llm`, is initialized with:
  - The `model_id` to specify which model to use.
  - The service `url` to connect to IBM watsonx.ai.
  - The `project_id` to associate the task with a specific project.
  - The `parameters` to configure the model's behavior.

##### 5. Invoking the model:

- The `granite_llm.invoke()` method is used to send a query to the model: "How to read a book effectively?".
- The model processes the input based on the parameters and returns a generated response.

## 6. Displaying the output:

- The response generated by the model is printed to the console using `print(response)`. This demonstrates how the model can be used to generate actionable insights or answers to user queries.

```
from ibm_watsonx_ai.foundation_models.utils.enums import ModelTypes # For specifying model types
from ibm_watsonx_ai import APIClient, Credentials # For API client and credentials management
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams # For managing model parameters
from ibm_watsonx_ai.foundation_models.utils.enums import DecodingMethods # For defining decoding methods
from langchain_ibm import WatsonxLLM, WatsonxEmbeddings # For interacting with IBM's LLM and embeddings
from ibm_watsonx_ai.foundation_models.utils import get_embedding_model_specs # For retrieving model specifications
from ibm_watsonx_ai.foundation_models.utils.enums import EmbeddingTypes # For specifying types of embeddings
from langchain.chains import LLMChain # For creating chains of operations with LLMs
from langchain.prompts import PromptTemplate # For defining prompt templates
parameters = {
    GenParams.DECODING_METHOD: "sample",
    GenParams.MAX_NEW_TOKENS: 512,
    GenParams.MIN_NEW_TOKENS: 1,
    GenParams.TEMPERATURE: 0.5,
    GenParams.TOP_K: 50,
    GenParams.TOP_P: 1,
}
# Set the new model ID for granite-3-3-8b-instruct
model_id = 'ibm/granite-3-3-8b-instruct'
# Define the project ID
project_id = "skills-network"
# Initialize the model with the new model_id
granite_llm = WatsonxLLM(
    model_id=model_id,
    url="https://us-south.ml.cloud.ibm.com",
    project_id=project_id,
    params=parameters,
)
response = granite_llm.invoke("How to read a book effectively?")
print(response)
```

You can then run the script in the terminal by using the following command.

```
python3 simple_llm.py
```

Upon running the script, you should see the generated text in your terminal, as shown in the following image. You see how watsonx Granite 3.0 provides a good answer. Note that the response you get might differ.

## Step 4. Clean up the transcript

Next, we define helper functions to clean up the transcript. The explanations in this section will help you understand the code and its purpose. **There's no need to copy-paste them anywhere for now.**

### Code explanation and purpose

#### 1. Function to remove non-ASCII characters

This function removes all characters outside the ASCII range from a given text. ASCII characters include standard English letters, numbers, and symbols. It ensures compatibility with systems or models that only process ASCII characters and helps avoid errors caused by unsupported symbols such as emojis or accented letters.

```
def remove_non_ascii(text):
    return ''.join(i for i in text if ord(i) < 128)
```

#### 2. Product assistant function

The `product_assistant` function processes transcripts of the sample earnings call to ensure that financial product terms are correctly formatted. It uses a detailed system prompt to transform terms (e.g., '401k' to '401(k) retirement savings plan') and resolves contextual ambiguities for acronyms such as 'LTV'. Note that in this project, this assistant is specifically designed to work with financial terminology. However, you can customize its prompts to suit your unique use case.

It concatenates the user input with the prompt and sends it to a language model (`meta-llama/llama-3-2-11b-vision-instruct`) using the `ModelInference` class. [Llama 3.2](#) collection represents a new generation of multimodal models that build upon the foundation established by Llama 3.1. Parameters such as `temperature` (randomness) and `top_p` (output diversity) customize the model's response. The function returns the formatted transcript after the model processes the input.

```
def product_assistant(ascii_transcript):
    system_prompt = """You are an intelligent assistant specializing in financial products;
    your task is to process transcripts of earnings calls, ensuring that all references to
    financial products and common financial terms are in the correct format. For each
    financial product or common term that is typically abbreviated as an acronym, the full term
    should be spelled out followed by the acronym in parentheses. For example, '401k' should be
    transformed to '401(k) retirement savings plan', 'HSA' should be transformed to 'Health Savings Account (HSA)', 'ROA' should be
    transformed to 'Return on Assets'.
    # Concatenate the system prompt and the user transcript
    prompt_input = system_prompt + "\n" + ascii_transcript
    # Create a messages object
    messages = [
        {
            "role": "user",
            "content": prompt_input
        }
    ]
    # Construct the model ID using the specified model size
    model_id = f"meta-llama/llama-3-2-11b-vision-instruct"
    # Configure the parameters for model behavior
    params = TextChatParameters(
        temperature=0.2, # Controls randomness; lower values make the output more deterministic
        top_p=0.6 # Nucleus sampling to control output diversity
    )
    # Initialize the Llama 3.2 model inference object
    llama32 = ModelInference(
        model_id=model_id, # Specify the model ID to use (Llama 3.2)
        credentials=credentials, # Authentication credentials for accessing the model
        project_id=project_id, # Link to the associated project ID
        params=params # Parameters that define the model's response behavior
    )
    # Send the input messages to the model and retrieve its response
    response = llama32.chat(messages=messages)
    # Extract and return the content of the model's first response choice
    return response['choices'][0]['message']['content']
```

## Step 5. Add PromptTemplate and Chain

Next, we will create a `PromptTemplate` and `Chain` to structure the input for the LLM. The explanations in this section will help you understand the code and its purpose. You don't need to copy-paste them anywhere just yet.

### What is PromptTemplate?

A [prompt template](#) is a predefined structure or format used to guide the input provided to a language model. It organizes information and defines the desired output, ensuring that the model generates relevant and consistent responses. In this example, the prompt template asks the model to generate meeting minutes and a list of tasks based on a given context. It specifies key sections, such as "Meeting Minutes" and "Task List", to standardize the output format.

### What is a Chain?

A [chain](#) is a sequence of steps that processes data and interacts with a language model to produce the desired output. Each step in the chain performs a specific function, such as:

- Passing input data
- Applying the prompt template
- Invoking the language model and
- Parsing the output into a usable form.

The chain ensures these steps are executed in the correct order, streamlining the entire workflow.

### Why do we need them together?

The prompt template and chain work together to create an efficient and automated workflow:

- **Prompt template:** Provides a structured format for the model's input and ensures that the output aligns with the desired requirements.
- **Chain:** Automates the process by linking raw input (e.g., the transcript) to the model through the template and handling the output.

### How it works in this example:

1. **Context as input:** The transcript is passed as context using `RunnablePassthrough`, allowing the raw text to move through the chain.
2. **Applying the template:** The template structures the input, instructing the model to generate meeting minutes and tasks.

3. **Invoking the model:** The language model (llm) processes the formatted input and generates the output.  
 4. **Parsing the output:** The StrOutputParser ensures the output is formatted and ready for further use.

```
# Define the prompt template
template = """
Generate meeting minutes and a list of tasks based on the provided context.
Context:
{context}
Meeting Minutes:
- Key points discussed
- Decisions made
Task List:
- Actionable items with assignees and deadlines
"""
prompt = ChatPromptTemplate.from_template(template)
# Define the chain
chain = (
    {"context": RunnablePassthrough()} # Pass the transcript as context
    | prompt
    | llm
    | StrOutputParser()
)
```

This combination ensures tasks such as summarization and task generation are handled efficiently, accurately, and in a structured manner.

## Step 6. Put them all together

Create new Python file and call it `speech_analyzer.py`. You can click the below button to do so.

[Open `speech\_analyzer.py` in IDE](#)

In this step, you set up an LLM instance using the **IBM Granite Instruct model**, a powerful language model designed for instruction-based tasks. Then, you establish a **prompt template**, which serves as a structured guide to organize input and specify the desired format for the LLM's output. Prompt templates are essential for ensuring consistent and meaningful results from language models. You can learn more about this in the [LangChain prompt template documentation](#).

Next, you create a **transcription function** that uses the **OpenAI Whisper model** to convert speech from audio files into text. This function processes audio files uploaded through a **Gradio app interface** (commonly in `.mp3` or `.wav` format, but other formats may also work). Once the transcription is complete, the resulting text is passed into an **LLMChain**. The LLMChain integrates the transcribed text with the defined prompt template and forwards it to the chosen LLM (IBM Granite Instruct model).

Finally, the LLM generates a response based on the input and template, and the output is displayed in the **Gradio app's output textbox**, providing a seamless end-to-end workflow for processing audio inputs and generating structured meeting minutes and tasklist in text format available for download. The tasklist provides actionable insights derived from the transcript, including additional information not explicitly stated in it. The LLM will help identify the next steps to take, and the user will then assess whether these tasks are feasible.

### Exercise: Copy paste the code below to your `speech_analyzer.py` file and fill in the missing parts.

```
import torch
import os
import gradio as gr
from ibm_watsonx_ai.foundation_models.utils.enums import ModelTypes
from ibm_watsonx_ai import APIClient
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams
from ibm_watsonx_ai.foundation_models.utils.enums import DecodingMethods
from ibm_watsonx_ai.foundation_models.schema import TextChatParameters
from ibm_watsonx_ai.foundation_models import ModelInference
from langchain_ibm import WatsonxLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain.prompts import PromptTemplate
from transformers import pipeline # For Speech-to-Text
#####
# IBM Watsonx LLM credentials
project_id = "skills-network"
credentials = Credentials(
    url="https://us-south.ml.cloud.ibm.com",
    # api_key="" # Normally you'd put an API key here
)
client = APIClient(credentials)
model_id = "ibm/granite-3-3-8b-instruct"
parameters = {
    GenParams.DECODING_METHOD: "sample",
    GenParams.MAX_NEW_TOKENS: 512,
    GenParams.MIN_NEW_TOKENS: 1,
    GenParams.TEMPERATURE: 0.5,
    GenParams.TOP_K: 50,
    GenParams.TOP_P: 1,
}
# Initialize IBM Watsonx LLM
llm = ## Your code here ##
#####
# Helper Functions#####
# Function to remove non-ASCII characters
```

```

def remove_non_ascii(text):
    return ''.join(i for i in text if ord(i) < 128)
def product_assistant(ascii_transcript):
    ## Your code here ##
#####
# Prompt Template and Chain-----
# Define the prompt template
template = """
Generate meeting minutes and a list of tasks based on the provided context.
Context:
{context}
Meeting Minutes:
- Key points discussed
- Decisions made
Task List:
- Actionable items with assignees and deadlines
"""

prompt = ChatPromptTemplate.from_template(template)
# Define the chain
chain = (
    ## Your code here ##
)
#####
# Speech-to-text pipeline-----
# Speech-to-text pipeline
def transcript_audio(audio_file):
    pipe = pipeline(
        ## Your code here - Please use model="openai/whisper-medium" ##
    )
    raw_transcript = pipe(audio_file, batch_size=8)["text"]
    ascii_transcript = remove_non_ascii(raw_transcript)
    adjusted_transcript = product_assistant(ascii_transcript)
    result = chain.invoke({"context": adjusted_transcript})
    # Write the result to a file for download
    output_file = "meeting_minutes_and_tasks.txt"
    with open(output_file, "w") as file:
        file.write(result)
    # Return the textual result and the file for download
    return result, output_file
#####
# Gradio Interface-----
audio_input = gr.Audio(sources="upload", type="filepath", label="Upload your audio file")
output_text = gr.Textbox(label="Meeting Minutes and Tasks")
download_file = gr.File(label="Download the Generated Meeting Minutes and Tasks")
iface = gr.Interface(
    fn=transcript_audio,
    inputs=audio_input,
    outputs=[output_text, download_file],
    title="AI Meeting Assistant",
    description="Upload an audio file of a meeting. This tool will transcribe the audio, fix product-related terminology, and generate meeting minutes and tasks."
)
iface.launch(server_name="0.0.0.0", server_port=5000)

```

► Click for the solution

Once complete, try running your code here.

```
python3 speech_analyzer.py
```

If there is no error, run the web app by clicking the following button. **Please note that if you want to download the output file, you have to run the application in a new tab, outside of the CloudIDE environment.**

[Web Application](#)

**Note:** If this "Web Application" button does not work, follow the following picture instructions to launch the application.

The output should look like the following example. Upload an audio file and then click **Submit** to generate meeting minutes.

Notice how the LLM corrected a minor mistake made by the speech-to-text model, resulting in a coherent and accurate output.

---

**Note:** Occasionally, the CloudIDE may go offline after a file upload. If this happens, wait for 2-3 minutes for the IDE to come back online, then click **Submit**. If it takes longer, try refreshing your browser.

Press **Ctrl + C** to stop the application.

## Exercise: Try with a different LLM

In our application, we used `ibm/granite-3-8b-instruct` as the primary LLM to generate meeting summaries and task lists. Now, try replacing it with `meta-llama/llama-4-maverick-17b-128e-instruct-fp8` and compare their performance to see how the outputs differ!

► Click here for hints

## Exercise: Try different model parameters

Now, try tweaking the model parameters—such as temperature, top-k, top-p, and max tokens—to see how the outputs change. These adjustments can influence the tone, creativity, and length of the summaries and task lists, so it's worth experimenting to find the best configuration for your use case.

► Click here for hints

# Conclusion

## Congratulations on Completing This Project!

You've successfully built a solid foundation for leveraging powerful LLMs for speech-to-text and text-processing tasks. Let's recap what you accomplished:

- **Text generation with LLMs:**

You developed a Python script to generate text using an advanced model from IBM WatsonX.ai. You also explored key parameters that influence the model's output and learned how to switch between different LLMs.

- **Speech-to-text conversion:**

You leveraged OpenAI's Whisper model to accurately convert audio recordings, such as lectures or meetings, into text—showcasing the power of modern transcription tools.

- **Content summarization:**

By integrating IBM WatsonX.ai, you summarized transcribed content effectively, extracting key points and actionable tasks, making the information concise and easy to follow.

- **User interface development:**

You designed an intuitive, user-friendly interface using Gradio, ensuring the application is accessible and easy to use for users—students, professionals, or educators.

Well done on completing this hands-on project! You've gained valuable experience in integrating cutting-edge AI tools into practical applications. Happy Learning!

---

## About the authors

[Hailey Quach](#) is a Data Scientist at IBM. She's completing her BSc, Honors in Computer Science at Concordia University, Montreal.