

Babel Fish with LLM STT TTS



Estimated time needed: 90 minutes

Introduction

Welcome to this guided project on creating a Babel Fish. Babel Fish is a metaphor for a translation service or a tool - based on a fictional creature from Douglas Adams' "The Hitchhiker's Guide to the Galaxy" series.

In this project, you will create a voice translation assistant using Watsonx and IBM Watson Speech Libraries for Embed. The guided project takes you through building a translation assistant that can take voice input, convert it to text using speech-to-text technology, send the text to Watsonx's mistralai/mistral-large, receive a response in other languages, convert it to speech using text-to-speech technology and finally play it back to the user. The voice translator will have a responsive frontend using HTML, CSS, and JavaScript, and a reliable backend using Flask.

Click [here](#) to play with a demo of the final application that you will create!

By the end of the course, you will have the skills to create your own AI-powered translation assistant capable of receiving voice input and providing translations in various languages. You will also have a solid foundation in web development using Python, Flask, HTML, CSS, and JavaScript, along with a fully functional full-stack application that is certain to impress anyone who interacts with it!

Before we begin, let's give some context of each topic.

Watsonx

Watsonx is an AI and data platform with a set of AI assistants designed to help you scale and accelerate the impact of AI with trusted data across your business.

The core components include a studio for new foundation models, generative AI, and machine learning; a fit-for-purpose data store built on an open data lakehouse architecture; and a toolkit, to accelerate AI workflows that are built with responsibility, transparency, and explainability.

The Watsonx AI assistants empower individuals in your organization to do work without expert knowledge across a variety of business processes and applications, including automating customer service, generating code, and automating key workflows in departments such as HR.

IBM Watson Speech Libraries for Embed

IBM Watson® Speech Libraries for Embed is a set of containerized text-to-speech and speech-to-text libraries designed to offer our IBM partners greater flexibility to infuse the best of IBM Research® technology into their solutions. Now available as embeddable AI, partners gain greater capabilities to build voice transcription and voice synthesis applications more quickly and deploy them in any hybrid multi-cloud environment. These technologies allow the assistant to communicate with users through voice input and output.

Voice assistants

A virtual assistant is a program designed to simulate conversation with human users, especially over the Internet using natural human voice. Assistants can be used in a variety of industries, including customer service, e-commerce, and education.

Python (Flask)

Python is a popular programming language that is widely used in web development and data science. Flask is a web framework for Python that makes it easy to build web applications. We will be using Python and Flask to build the backend of our voice assistant. Python is a powerful language that is easy to learn and has a large ecosystem of libraries and frameworks that can be leveraged in projects like ours.

HTML - CSS - JavaScript

HTML (Hypertext Markup Language) is a markup language used to structure content on the web. CSS (Cascading Style Sheets) is a stylesheet language used to describe the look and formatting of a document written in HTML. JavaScript is a programming language that is commonly used to add interactivity to web pages. Together, these technologies allow us to build a visually appealing and interactive frontend for our assistant. Users will be able to interact with the voice assistant through a web interface that's built using HTML, CSS, and JavaScript.

Learning objectives

At the end of this project, you will be able to:

- Explain the basics of voice assistants and their various applications
- Set up a development environment for building an assistant using Python, Flask, HTML, CSS, and JavaScript
- Implement speech-to-text functionality to allow the assistant to understand voice input from users
- Integrate the assistant with Watsonx's mistralai/mistral-large model to give it a high level of intelligence and the ability to understand and respond to user requests
- Implement text-to-speech functionality to allow the assistant to communicate with users through voice output
- Combine all of the above components to create a functional assistant that can take voice input and provide a spoken response
- Deploy as web application by running a server in development mode

Prerequisites

For working on this project, you should have a fundamental knowledge of Python. In addition, having knowledge of the basics of HTML, CSS, and JavaScript are nice to have but not essential. We will do our best in explaining each step of the process as well as any code shown along the way.

Step 1: Understanding the interface

In this project, the goal is to create an interface that allows communication with a voice assistant, and a backend to manage the sending and receiving of responses.

The frontend will use HTML, CSS and JavaScript with popular libraries such as Bootstrap for basic styling, Font Awesome for icons and JQuery for efficient handling of actions. The user interface will be similar to other voice assistant applications, like Google Assistant. The code for the interface is provided and the focus of the course is on building the voice assistant and integrating it with various services and APIs. The provided code will help you to understand how the frontend and backend interact, and as you go through it, you will learn about the important parts and how it works, giving you a good understanding of how the frontend works and how to create this simple web page.

First, let's set up the environment by executing the following code:

```
python3.11 -m venv my_env
source my_env/bin/activate # activate my_env
```

Run the following commands in the terminal to receive the outline of the project, rename it with another name and finally move into that directory:

```
git clone https://github.com/ibm-developer-skills-network/translator-with-voice-and-watsonx
cd translator-with-voice-and-watsonx
```

installing the requirements for the project

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

Have a cup of coffee, it will take 5-10 minutes to install the requirements (You can continue this project while the requirements are installed).

```
) (
( ) )
) ( (
) )
_ _ _ _ _
|
| C | / \ / \ / \ / \
| ' - ' / \ / \ / \ / \
| _ _ _ _ _
| _ _ _ _ _
```

The next section gives a brief understanding of how the frontend works.

HTML, CSS, and JavaScript

The `index.html` file is responsible for the layout and structure of the web interface. This file contains the code for incorporating external libraries such as JQuery, Bootstrap, and FontAwesome Icons, as well as the CSS (`style.css`) and JavaScript code (`script.js`) that control the styling and interactivity of the interface.

The `style.css` file is responsible for customizing the visual appearance of the page's components. It also handles the loading animation using CSS keyframes. Keyframes are a way of defining the values of an animation at various points in time, allowing for a smooth transition between different styles and creating dynamic animations.

The `script.js` file is responsible for the page's interactivity and functionality. It contains the majority of the code and handles all the necessary functions such as switching between light and dark mode, sending messages, and displaying new messages on the screen. It even enables the users to record audio.

Images of UI

Here are some images of the frontend you received.

Light mode

This image demonstrates how the base code works. It'll just return `null` as a response.

Your personal assistant powered by

Try asking: What can you sp

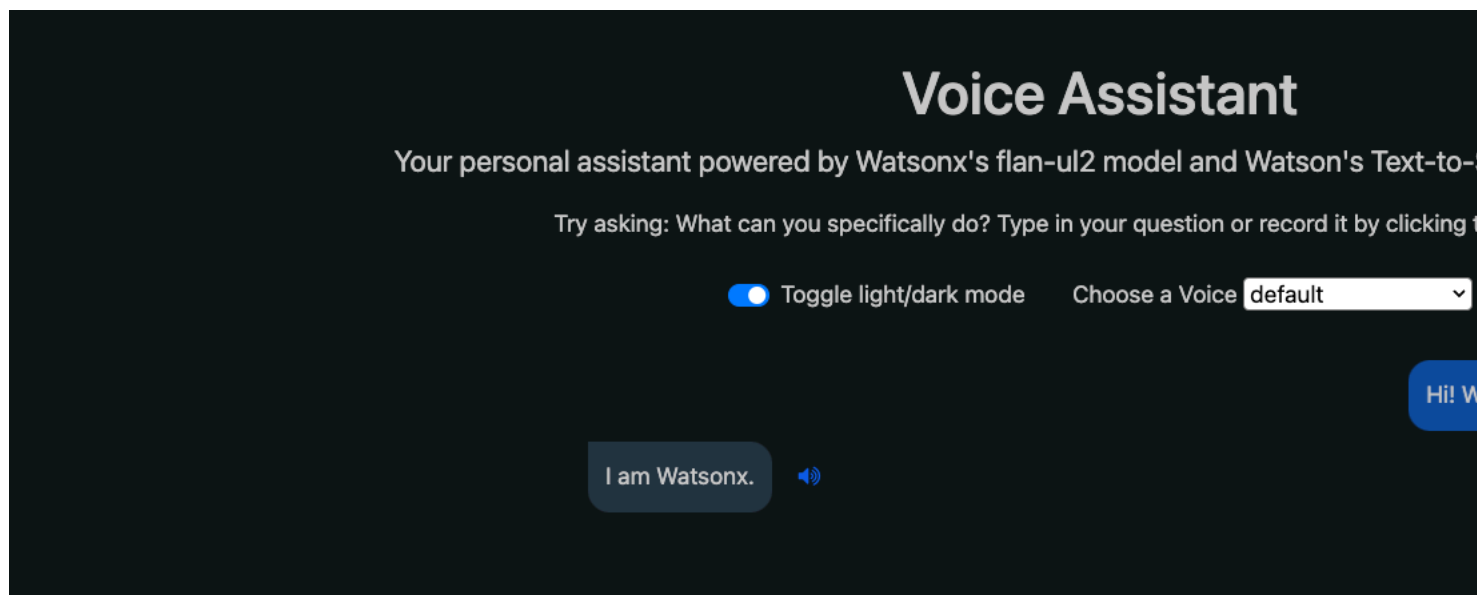


Toggle



Dark mode

Once you go through the project, you'll complete the assistant and it will be able to give clear responses as shown below:



Step 2: Understanding the server

The server is how the application will run and communicate with all our services. Flask is a web development framework for Python and can be used as a backend for the application. It is a lightweight and simple framework that makes it quick and easy to build web applications.

With Flask, you can create web pages and applications without requiring a lot of complex coding or using additional tools or libraries. You can create your own routes and handle user requests, and it also allows you to connect to external APIs and services to retrieve or send data.

This guided project uses Flask to handle the backend of your voice assistant. This means that you will be using Flask to create routes and handle HTTP requests and responses. When a user interacts with the voice assistant through the frontend interface, the request will be sent to the Flask backend. Flask will then process the request and send it to the appropriate service.

The code provided gives the outline for the server in the `server.py` file.

[Open `server.py` in IDE](#)

At the top of the file, there are several import statements. These statements are used to bring in external libraries and modules, which will be used in the current file. For example, `speech_text` is a function inside the `worker.py` file, while `ibm_watson_machine_learning` is a package that needs to be installed to use Watsonx's `mistralai/mistral-large` model. These imported packages, modules, and libraries will allow you to access the additional functionalities and methods that they offer, making it easy to interact with the speech-to-text and `mistralai/mistral-large` models in your code.

Underneath the imports, the Flask application is initialized, and a CORS policy is set. A CORS policy is used to allow or prevent web pages from making requests to different domains than the one that served the web page. Currently, it is set to `*` to allow any request.

The `server.py` file consists of 3 functions which are defined as routes, and the code to start the server.

Replace the first route in the `server.py` with the code below:

```
@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')
```

Function explanation

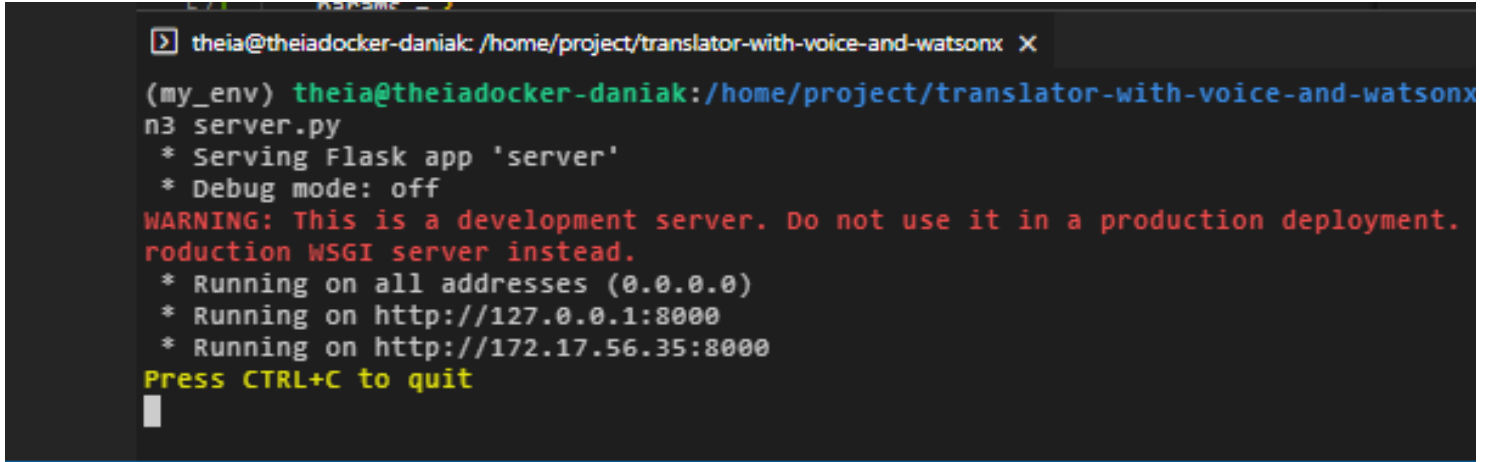
When a user tries to load the application, they initially send a request to go to the `/` endpoint. They will then trigger this `index` function and execute the code above. Currently, the returned code from the function is a render function to show the `index.html` file which is the frontend interface.

The second and third routes will be used to process all requests and handle sending information between the applications.

Now you just need to run the `server.py` file and start the application.

```
python server.py
```

You will have the following output in the terminal. This shows the server is running.



```

theia@theiadocker-daniak: /home/project/translator-with-voice-and-watsonx X
(my_env) theia@theiadocker-daniak:/home/project/translator-with-voice-and-watsonx
n3 server.py
* Serving Flask app 'server'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
roduction WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:8000
* Running on http://172.17.56.35:8000
Press CTRL+C to quit

```

The app is now running on localhost:8000. To access the application click the following button:

Launch the application

(If you are running the file locally, you can open the browser and copy the running URL, in this case, it is `http://127.0.0.1:8000`.)

The application interface will open. At this point, the application will run but will not return any output. Once you've had a chance to run and play around with the application, please press `Ctrl` (a.k.a. `control` (^) for Mac) and `C` at the same time in the terminal to stop the application.

The next sections will take you through the process of completing the `process_message_route` and `speech_to_text_route` functions in this file and help you understand how to use the packages and endpoints.

Step 3: Integrating Watsonx API

It's time to give your voice assistant a brain! With the power of Watsonx's API, we can pass the transcribed text and receive responses that answer your questions.

Authenticating for programmatic access

In this project, you do not need to specify your own `Watsonx_API` and `Project_id` to the below `worker.py` code. You can just specify `project_id="skills-network"` and leave `Watsonx_API` blank, as in this CloudIDE environment, we have already granted you access to API without your own `Watsonx_API` and `Project_id`.

But it's important to note that this access method is exclusive to this Cloud IDE environment. If you are interested in using the model/API outside this environment (for example, in a local environment), detailed instructions and further information are available in this [tutorial](#).

Open `worker.py` in IDE

Add the following at the top of the file:

```

# To call watsonx's LLM, we need to import the library of IBM Watson Machine Learning
from ibm_watson_machine_learning.foundation_models.utils.enums import ModelTypes
from ibm_watson_machine_learning.foundation_models import Model
# placeholder for Watsonx_API and Project_id incase you need to use the code outside this environment
# API_KEY = "Your WatsonX API"
PROJECT_ID= "skills-network"
# Define the credentials
credentials = {
    "url": "https://us-south.ml.cloud.ibm.com"
    #"apikey": API_KEY
}

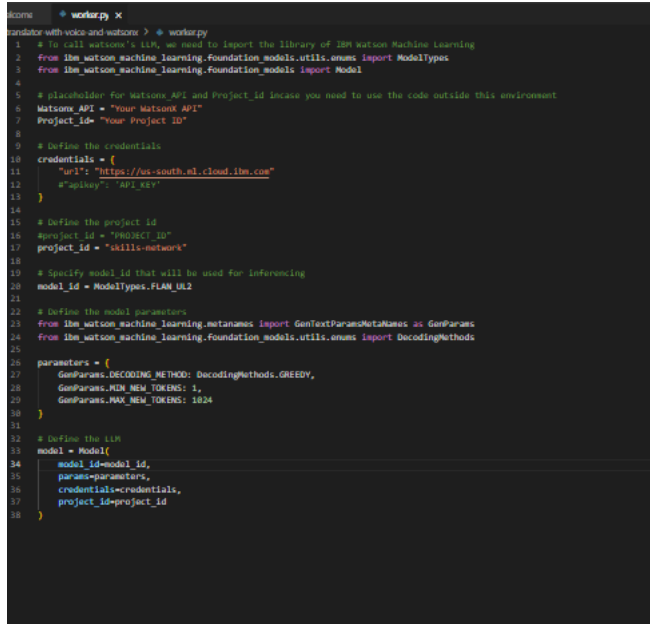
# Specify model_id that will be used for inferencing
model_id = "mistralai/mistral-medium-2505"
# Define the model parameters
from ibm_watson_machine_learning.metanames import GenTextParamsMetaNames as GenParams
from ibm_watson_machine_learning.foundation_models.utils.enums import DecodingMethods
parameters = {
    GenParams.DECODING_METHOD: DecodingMethods.GREEDY,
    GenParams.MIN_NEW_TOKENS: 1,
    GenParams.MAX_NEW_TOKENS: 1024
}
# Define the LLM

```

```

model = Model(
    model_id=model_id,
    params=parameters,
    credentials=credentials,
    project_id=PROJECT_ID
)

```



```

# Import the libraries
# To call Watsonx's LLM, we need to import the library of IBM Watson Machine Learning
from ibm_watson_machine_learning.foundation_models.utils.enums import ModelTypes
from ibm_watson_machine_learning.foundation_models import Model

# Placeholder for Watsonx API and Project ID because you need to use the code outside this environment
Watsonx_API = "Your Watsonx API"
Project_ID = "Your Project ID"

# Define the credentials
credentials = {
    "url": "https://us-south.ml.cloud.ibm.com",
    "apikey": "API KEY"
}

# Define the project ID
project_id = "PROJECT ID"

# Specify model id that will be used for inferencing
model_id = ModelTypes.FLAN_102

# Define the model parameters
from ibm_watson_machine_learning.metanames import GenTextParamsMetaNames as GenParams
from ibm_watson_machine_learning.foundation_models.utils.enums import DecodingMethods

parameters = {
    GenParams.DECODING_METHOD: DecodingMethods.GREEDY,
    GenParams.MAX_NEW_TOKENS: 1,
    GenParams.MAX_NEW_TOKENS: 1024
}

# Define the LLM
model = Model(
    model_id=model_id,
    params=parameters,
    credentials=credentials,
    project_id=project_id
)

```

Watsonx process message function

We will be updating the function called `watsonx_process_message`, which will take in a prompt and pass it to Watsonx's `mistralai/mistral-medium-2505` API to receive a response. Essentially, it's the equivalent of pressing the send button to get a response from ChatGPT.

Go ahead and update the `watsonx_process_message` function in the `worker.py` file with the following.

```

def watsonx_process_message(user_message):
    # Set the prompt for Watsonx API - using a strict translation instruction
    prompt = f"Respond to the query: ```{user_message}```"
    response_text = model.generate_text(prompt=prompt)
    print("watsonx response:", response_text)
    return response_text.strip()

```

Prompt refinement

We will have to optimize our translation assistant. Since this is a translator, users shouldn't have to type "translate" every time. To address this, we've improved the prompt in the `watsonx_process_message` function to be more explicit.

For example, we now focus on translating sentences from English into Spanish, the updated prompt will look like below. Replace the prompt in the function with this:

```

prompt = f"""
Translate the following English sentence into Spanish.
Reply ONLY with the translation, no explanations, no formatting, no extra text.
English: {user_message}
Spanish:
"""

```

This revised prompt makes it evident that the user intends to translate a sentence into Spanish, eliminating the need to explicitly mention "translate."

If your translation needs to involve languages other than Spanish, you can easily adapt the prompt. Simply replace "Spanish" in the prompt with the name of your required target language. This modification simplifies the user interaction and ensures that the translator remains user-friendly for various language pairs.

Function explanation

The function is really simple, thanks to the very easy-to-use `ibm_watson_machine_learning` library.

Then we call Watsonx's API by using `model.generate_text` function and pass the prompt that we need the response for. Remember that `model` refers to the LLM we established earlier.

Again, you can tweak these parameters according to your personalized needs and can learn more about them by going to [IBM Watsonx Prompt Lab](#) where you can test all parameters in real-time like below:

Finally, we return the `response_text` which stores the answer to our prompt.

Step 4: Integrating Watson Speech-to-Text

Speech-to-Text functionality is a technology that converts speech into text using machine learning. It is useful for accessibility, productivity, convenience, multilingual support, and cost-effective solutions for a wide range of applications. For example, being able to take a user's voice as input for a chat application.

Using the embedded Watson Speech-to-Text AI model that was deployed earlier, it is possible to easily convert our Speech-to-Text by a simple API. This result can then be passed to Watsonx API for generating a response.

Starting Speech-to-Text

Skills Network provides its own Watson Speech-to-Text image that is run automatically in this environment. To access it, use this endpoint URL: `https://sn-watson-stt.labs.skills.network`

You can test it works by running this query:

```
curl https://sn-watson-stt.labs.skills.network/speech-to-text/api/v1/models
```

You should see a list of a few languages it can recognize. An example output is shown below.

```
{
  "models": [
    {
      "name": "es-LA_Telephony",
      "language": "es-LA",
      "description": "Latin American Spanish telephony model for narrowband audio (8kHz)",
      ...
    },
    {
      "name": "en-US_Multimedia",
      "language": "en-US",
      "description": "US English multimedia model for broadband audio (16kHz or more)",
      ...
    }
  ]
}
```

Next, try getting an example audio file to send a /recognize request to test the service. For example, you can download the example audio file by this command:

```
curl "https://github.com/watson-developer-cloud/doc-tutorial-downloads/raw/master/speech-to-text/0001.flac" -sLo example.flac
```

Send the audio file to the service:

```
curl "https://sn-watson-stt.labs.skills.network/speech-to-text/api/v1/recognize" --header "Content-Type: audio/flac" --data-binary @example.flac
```

Example response:

```
{
  "result_index": 0,
  "results": [
    {
      "final": true,
      "alternatives": [
        {
          "transcript": "several tornadoes touched down as a line of severe thunderstorms swept through colorado on sunday ",
          "confidence": 0.99
        }
      ]
    }
  ]
}
```

To use a different model, add the model query parameter to the request. The audio format can also be changed as long as the Content-Type header matches. For example:

```
curl "https://sn-watson-stt.labs.skills.network/speech-to-text/api/v1/recognize?model=es-LA_Telephony" --header "Content-Type: audio/flac" --data-bir
{
  "result_index": 0,
  "results": [
    {
      "final": true,
      "alternatives": [
        {
          "transcript": "s ",
          "confidence": 0.39
        }
      ]
    }
  ]
}
```

Implementation

We will be updating a function called `speech_to_text` in the `worker.py` file that will take in audio data received from the browser and pass it to the Watson Speech-to-Text API.

The `speech_to_text` function will take in audio data as a parameter, make an API call to the Watson Speech-to-Text API using the `requests` library, and return the transcription of the audio data.

Remember to replace the ... for the `base_url` variable with the URL for your Speech-to-Text model (for example, `https://sn-watson-stt.labs.skills.network`).

Open `worker.py` in IDE

```
import requests
def speech_to_text(audio_binary):
    # Set up Watson Speech-to-Text HTTP Api url
    base_url = '...'
    api_url = base_url + '/speech-to-text/api/v1/recognize'
    # Set up parameters for our HTTP request
    params = {
        'model': 'en-US_Multimedia',
    }
    # Set up the body of our HTTP request
    body = audio_binary
    # Send a HTTP Post request
    response = requests.post(api_url, params=params, data=audio_binary).json()
    # Parse the response to get our transcribed text
    text = 'null'
    while bool(response.get('results')):
        print('Speech-to-Text response:', response)
        text = response.get('results').pop().get('alternatives').pop().get('transcript')
        print('recognised text: ', text)
    return text
```

Function explanation

The `requests` library imported at the top of our `worker.py` file is a simple HTTP request library that we will be using to make API calls to the Watson Speech-to-Text API.

The function simply takes `audio_binary` as the only parameter and then sends it in the body of the HTTP request.

To make an HTTP Post request to Watson Speech-to-Text API, we need the following three elements:

1. **URL** of the API: This is defined as `api_url` in our code and points to Watson's Speech-to-Text service
2. **Parameters**: This is defined as `params` in our code. It's just a dictionary having one key-value pair i.e. `'model': 'en-US_Multimedia'` which tells Watson that we want to use the US English model for processing our speech
3. **Body** of the request: this is defined as `body` and is equal to `audio_binary` since we are sending the audio data inside the body of our POST request.

We then use the `requests` library to send this HTTP request passing in the URL, params, and data(body) to it and then use `.json()` to convert the API's response to json format which is very easy to parse and can be treated like a dictionary in Python.

The structure of the response is something like this:

```
{
  "response": {
    "results": {
      "alternatives": {
        "transcript": "Recognised text from your speech"
      }
    }
  }
}
```

Therefore, we check if the response contains any results, and if it does, we extract the text by getting the nested transcript string as shown above. Then return this text.

Small tip

Notice the print statements such as `print('response', response)`, it's always a good idea to print out the data you are receiving from some external place like an API in this case, as it really helps with debugging if something goes wrong.

Step 5: Integrating Watson Text-to-Speech

Time to give your assistant a voice using Text-to-Speech functionality.

Once we have processed the user's message using Watsonx, let's add the final worker function that will convert that response to speech, so you get a more personalized feel as the Personal Assistant is going to read out the response to you. Just like other virtual assistants like Google, Alexa, Siri, etc.

Starting Text-to-Speech

Skills Network provides its own Watson Text-to-Speech image that is run automatically in this environment. To access it, use this endpoint URL: <https://sn-watson-tts.labs.skills.network>

You can test it works by running this query:

```
curl https://sn-watson-tts.labs.skills.network/text-to-speech/api/v1/voices
```

You should see a list of a bunch of different voices this model can use. An example output is shown below.

```
{
  "voices": [
    {
      "name": "en-US_OliviaV3Voice",
      "language": "en-US",
      "gender": "female",
      "description": "Olivia: American English female voice. Dnn technology.",
      ...
    },
    {
      "name": "es-ES_EnriqueV3Voice",
      "language": "en-GB",
      "gender": "male",
      "description": "Enrique: Castilian Spanish (español castellano) male voice. Dnn technology.",
      ...
    },
    ...
  ]
}
```

Next, try sending an example text (ex: "Hello world") in JSON format to invoke /synthesize request. It will return an audio file named "output.wav" in the "translator-with-voice-and-watsonx" directory:

```
curl "https://sn-watson-stt.labs.skills.network/text-to-speech/api/v1/synthesize" --header "Content-Type: application/json" --data '{"text":"Hello wc
```

To use a different model, add the voice query parameter to the request. To change the audio format, change the Accept header. For example:

```
curl "https://sn-watson-stt.labs.skills.network/text-to-speech/api/v1/synthesize?voice=es-LA_SofiaV3Voice" --header "Content-Type: application/json"
```

After executing the above command, you'll find the output file named "hola.mp3" in the "translator-with-voice-and-watsonx" directory.

Text-to-Speech function

In the `worker.py` file, the `text_to_speech` function passes data to Watson's Text-to-Speech API to get the data as spoken output.

This function is going to be similar to `speech_to_text` as we will be utilizing our request library again to make an HTTP request. Lets dive into the code. **Again, remember to replace the ... for the `base_url` variable with the URL for your Text-to-Speech model (for example, `https://sn-watson-tts.labs.skills.network`).**

Open `worker.py` in IDE

```
def text_to_speech(text, voice=""):
    # Set up Watson Text-to-Speech HTTP Api url
    base_url = '...'
    api_url = base_url + '/text-to-speech/api/v1/synthesize?output=output_text.wav'
    # Adding voice parameter in api_url if the user has selected a preferred voice
    if voice != "" and voice != "default":
        api_url += "&voice=" + voice
    # Set the headers for our HTTP request
    headers = {
        'Accept': 'audio/wav',
        'Content-Type': 'application/json',
    }
    # Set the body of our HTTP request
    json_data = {
        'text': text,
    }
    # Send a HTTP Post request to Watson Text-to-Speech Service
    response = requests.post(api_url, headers=headers, json=json_data)
    print('Text-to-Speech response:', response)
    return response.content
```

Function explanation

The function takes `text` and `voice` as parameters. It adds `voice` as a parameter to the `api_url` if it's not empty or not default. It sends the `text` in the body of the HTTP request.

Similarly as before, to make an HTTP Post request to Watson Text-to-Speech API, we need the following three elements:

1. **URL** of the API: This is defined as `api_url` in our code and points to Watson's Text-to-Speech service. This time we also append a `voice` parameter to the `api_url` if the user has sent a preferred voice in their request.
2. **Headers**: This is defined as `headers` in our code. It's just a dictionary having two key-value pairs. The first is `'Accept': 'audio/wav'` which tells Watson that we are sending audio having `wav` format. The second one is `'Content-Type': 'application/json'`, which means that the format of the body would be *JSON*.
3. **Body** of the request: This is defined as `json_data` and is a dictionary containing `'text': text` key-value pair, this text will then be processed and converted to a speech.

We then use the `requests` library to send this HTTP to request passing in the URL, headers, and `json(body)` to it and then use `.json()` to convert the API's response to `json` format so we can parse it.

The structure of the response is something like this:

```
{
  "response": {
    content: The Audio data for the processed Text-to-Speech
  }
}
```

Therefore, we return `response.content` which contains the audio data received.

Step 6: Putting everything together by creating Flask API endpoints

Now by using the functions we defined in the previous sections, we can connect everything and complete the assistant.

The changes in this section will be for the `server.py` file.

Open `server.py` in IDE

The outline has already taken care of the imports for the functions from the `worker.py` file to the `server.py` file. This allows the `server.py` file to access these imported functions from the `worker.py` file.

```
from worker import speech_to_text, text_to_speech, watsonx_process_message
```

Now we will be updating two Flask routes, one for converting the user's Speech-to-Text (`speech_to_text_route`) and the other for processing their message and converting the Watsonx's response back to speech (`process_message_route`).

Speech-to-Text route

This function is simple, as it converts the user's Speech-to-Text using the `speech_to_text` we defined in one of our previous sections and returns the response. Replace the `speech_to_text_route` function with the code below:

```
@app.route('/speech-to-text', methods=['POST'])
def speech_to_text_route():
    print("processing Speech-to-Text")
    audio_binary = request.data # Get the user's speech from their request
    text = speech_to_text(audio_binary) # Call speech_to_text function to transcribe the speech
    # Return the response to user in JSON format
    response = app.response_class(
        response=json.dumps({'text': text}),
        status=200,
        mimetype='application/json'
    )
    print(response)
    print(response.data)
    return response
```

Function explanation

We start by storing the `request.data` in a variable called `audio_binary`, as we are sending the binary data of audio in the body of the request from the frontend. Then we use our previously defined function `speech_to_text` and pass in the `audio_binary` as a parameter to it. We store the return value in a new variable called `text`.

As our frontend expects a JSON response, we create a json response by using the Flask's `app.response_class` function and passing in three arguments:

1. `response`: This is the actual data that we want to send in the body of our HTTP response. We will be using `json.dumps` function and will pass in a simple dictionary containing only one key-value pair - `'text': text`
2. `status`: This is the status code of the HTTP response; we will set it to 200 which essentially means the response is *OK* and that the request has succeeded.
3. `mimetype`: This is the format of our response which is more formally written as `'application/json'` in HTTP request/response.

We then return the response.

Process message route

This function will basically accept a user's message in text form with their preferred voice. It will then use our previously defined helper functions to call the Watsonx's API to process this prompt and then finally convert that response to text using Watson's Text-to-Speech API and then return this data back to the user. Replace the `process_message_route` function to the code below:

```
@app.route('/process-message', methods=['POST'])
def process_message_route():
    user_message = request.json['userMessage'] # Get user's message from their request
    print('user_message', user_message)
    voice = request.json['voice'] # Get user's preferred voice from their request
    print('voice', voice)
    # Call watsonx_process_message function to process the user's message and get a response back
    watsonx_response_text = watsonx_process_message(user_message)
    # Clean the response to remove any emptylines
    watsonx_response_text = os.linesep.join([s for s in watsonx_response_text.splitlines() if s])
    # Call our text_to_speech function to convert Watsonx Api's response to speech
    watsonx_response_speech = text_to_speech(watsonx_response_text, voice)
    # convert watsonx_response_speech to base64 string so it can be sent back in the JSON response
    watsonx_response_speech = base64.b64encode(watsonx_response_speech).decode('utf-8')
    # Send a JSON response back to the user containing their message's response both in text and speech formats
    response = app.response_class(
        response=json.dumps({"watsonxResponseText": watsonx_response_text, "watsonxResponseSpeech": watsonx_response_speech}),
        status=200,
        mimetype='application/json'
    )
    print(response)
```

```
return response
```

Function explanation

We will start by storing the user's message in `user_message` by using `request.json['userMessage']`. Similarly, we will also store the user's preferred voice in `voice` by using `request.json['voice']`.

We will then use the helper function we defined earlier to process this user's message by calling `watsonx_process_message(user_message)` and storing the response in `watsonx_response_text`. We will then clean this response to remove any empty lines by using a simple one-liner function in Python that is, `os.linesep.join([s for s in watsonx_response_text.splitlines() if s])`.

Once we have this response cleaned, we will now use another helper function we defined earlier to convert it to speech. Therefore, we will call `text_to_speech` and pass in the two required parameters which are `watsonx_response_text` and `voice`. We will store the function's return value in a variable called `watsonx_response_speech`.

As the `watsonx_response_speech` is a type of audio data, we can't directly send this inside a json as it can only store textual data. Therefore, we will be using something called "*base64 encoding*". We can convert any type of binary data to a textual representation by encoding the data in base64 format. Hence, we will simply use `base64.b64encode(watsonx_response_speech).decode('utf-8')` and store the result back to `watsonx_response_speech`.

Now we have everything ready for our response so finally we will be using the same `app.response_class` function and send in the three parameters required. The `status` and `mimetype` will be exactly the same as we defined them in our previous `speech_to_text_route`. In the response we will use `json.dumps` function as we did before and will pass in a dictionary as a parameter containing `"watsonxResponseText": watsonx_response_text` and `"watsonxResponseSpeech": watsonx_response_speech`.

We then return the response.

Step 7: Running the app in CloudIDE

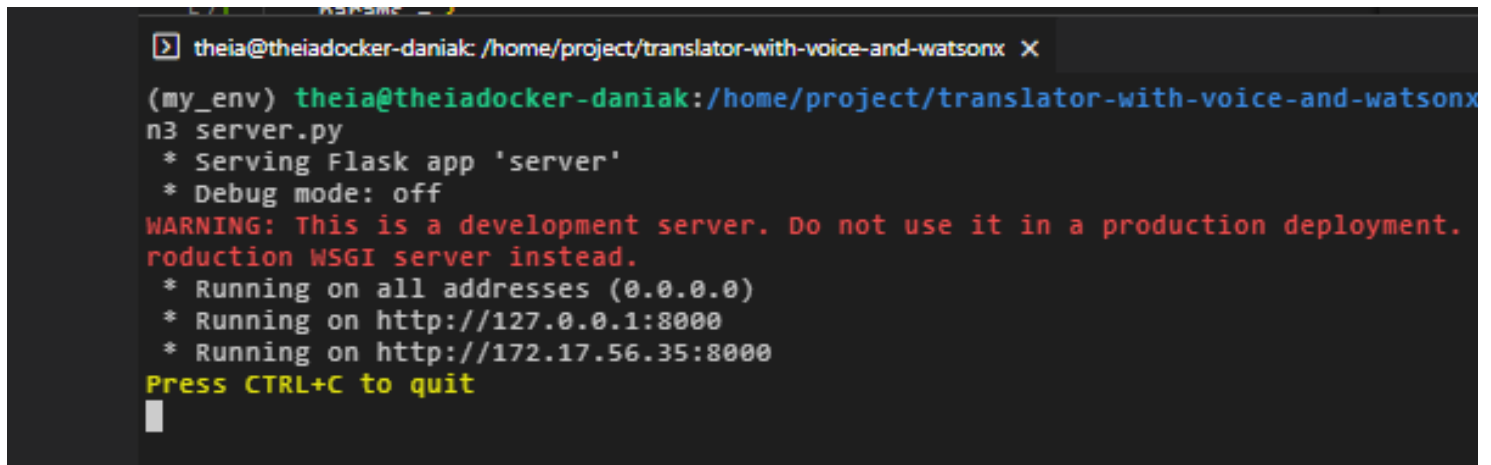
The assistant is now complete and ready to use.

Assuming the Text-to-Speech and Speech-to-Text models URLs are correctly set, you just need to run the `server.py` file and start the application.

You need to run the `server.py` file, first.

```
python server.py
```

You will have the following output in the terminal. This shows the server is running.



```
theia@theiadocker-daniak: /home/project/translator-with-voice-and-watsonx X
(my_env) theia@theiadocker-daniak:/home/project/translator-with-voice-and-watsonx
n3 server.py
* Serving Flask app 'server'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment.
roduction WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:8000
* Running on http://172.17.56.35:8000
Press CTRL+C to quit
```

The app is now running on localhost:8000. To access the application click the following button:

Launch the application

To stop the application, press `Ctrl+C` in the terminal.

(If you are running the file locally, you can open the browser and copy the running URL, in this case, it is `http://127.0.0.1:8000`.)

The application interface will open. Use the interface to test your application.

*Remember to test all the different inputs and features.
Most importantly switch the Voice to a **Spanish** one.
Also, try typing a message using the message box and using the microphone.*

Note: The browser may block the tab from opening up. Please enable for the application to work.

Step 9: Deploy to Code Engine (OPTIONAL)

IMPORTANT NOTE: WHEN DEPLOYING YOU MUST USE YOUR OWN API KEY AND WATSONX PROJECT ID

If you would like to host your application and have it available for anyone to use, you can follow these steps to deploy it. The deployment will be to IBM Cloud's Code Engine. IBM Cloud Code Engine is a fully managed, cloud-native service for running containerized workloads on IBM Cloud. It allows developers to deploy and run code in a secure, scalable, and serverless environment, without having to worry about the underlying infrastructure.

The following steps in Part 1 allow you to deploy to an IBM Skills Network Code Engine environment to test if everything is working just fine, which is deleted after a few days. Part 2 shows the steps to deploy for real to your own account.

Part 1: Deploying to Skills Network Code Engine

Step 1. Create Code Engine project

In the left-hand navigation panel, there is an option for the Skills Network Toolbox. Simply open that and expand the *CLOUD* section and then click on *Code Engine*. Finally, click on *Create Project*.

Step 2. Click on Code Engine CLI button

From the same page simply click on Code Engine CLI button. This will open a new terminal and will login to a code engine project with everything already set up for you.

Step 3. Deploy Speech-to-Text service

From the same terminal that opened in the last step, run the following command to deploy Watson Speech-to-Text Service:

```
ibmcloud ce application create --name speech-to-text \
--env ACCEPT_LICENSE=true \
--image us.icr.io/sn-labsassets/speech-standalone:latest \
--port 1080 \
--registry-secret icr-secret \
--min-scale 1 \
--visibility project
```

Parameter explanation:

- `--env` This argument allows you to pass in environment variables to your image. For the TTS and STT service we simply need to set `ACCEPT_LICENSE` to true.
- `--image` This specifies a pre-built image to run. Here you will use the skills network provided images.
- `--port` This lets Code Engine know what port the application runs on (it uses this for a health check). The TTS and STT service run on port `1080`.
- `--registry-secret` This is allowing you to pass in an Image Registry secret (for access to private container images). Here you will use `icr-secret` which has been pre-created for you by Skills Network (and has access to all the images).
- `--min-scale` Code Engine can automatically scale the servers that run your application (More demand = More servers). This value default to `0` which means that when the application is not in use it will shut down all the servers (and won't incur charges anymore). Here you'll set it to `1` to ensure we always have a service available.
- `--visibility` This allows you to set *who* is able to access your project. Here we set `project` as this service will *not* be public and only used by the application you coded privately.

Once the service is ready, it will output a URL in the terminal. Go to your `speech_to_text` function defined in the `worker.py` file, and then replace the `base_url` with that.

Step 4. Deploy Text-to-Speech service

From the same terminal window, run the following command to deploy Watson Text-to-Speech Service:

Similarly, once the service is ready, it will output a URL in the terminal. Simply go to your `text_to_speech` function defined in the `worker.py` file, and then replace the `base_url` with that.

```
ibmcloud ce application create --name text-to-speech \
--env ACCEPT_LICENSE=true \
--image us.icr.io/sn-labsassets/tts-standalone:latest \
--port 1080 \
--registry-secret icr-secret \
--min-scale 1 \
--visibility project
```

Step 5. Deploy your app

Finally, from the same terminal window (make sure you are in the `translator-with-voice-and-watsonx` directory), run the following command to deploy your app to Code Engine.

IMPORTANT NOTE: YOU MUST USE YOUR OWN API KEY AND WATSONX PROJECT ID

- Open `worker.py`
- Set both `API_KEY` and `PROJECT_ID`

```
ibmcloud ce application create --name personal-assistant \
--build-source . \
--build-context-dir . \
--image us.icr.io/${SN_ICR_NAMESPACE}/personal-assistant:latest \
--registry-secret icr-secret \
--port 8000 \
--min-scale 1 \
--visibility project
```

Parameter explanation (continued):

- `--build-source` This specifies where the code you want to build from exists. This can be a GitHub repository - in your case use `.` which just sets the current local directory.
- `--build-context-dir` This specifies from within the build source, where to find our Docker project (again we use `.`)
- `--image` This is similar to above, but this time we are specifying which image we want Code Engine to *create* for us.

Once the app is deployed, open the URL outputted in the terminal, and enjoy your app deployed live on the web thanks to Code Engine.

Part 2: Deploying to your own account!

The main difference between deploying to the Skills Network Environment and your own environment is that you need to deploy the root images for the Text-to-Speech and Speech-to-Text models yourself and use the correct endpoints.

Step 1. Log in to your IBM Cloud account.

Using the `ibmcloud login` command log into your own IBM Cloud account. Remember to replace `USERNAME` with your IBM Cloud account email and then enter your password when prompted to.

```
ibmcloud login -u USERNAME
```

Use `ibmcloud login --sso` command to log in, if you have a federated ID.

Then target any specific resource group in your account. By default, if you've completed the sign-up process for your IBM Cloud account, you can use the Default resource group.

```
ibmcloud target -g Default
```

Step 2. Login to the IBM Entitled Registry

You'll need to log in to the IBM Entitled Registry to download the Watson Speech-to-Text and Text-to-Speech images so you can deploy them to your own Code Engine project.

Go to [IBM's Container Library](#) to get an Entitlement Key. This key gives you access to pulling and using the IBM Watson Speech Libraries for Embed. However, do note that **this key is only valid for a Year as a trial**.

Once you've obtained the Entitlement Key from the container software library you can log in to the registry with the key, and pull the images.

Replace it with your own IBM Entitlement Key.

```
IBM_ENTITLEMENT_KEY="YOUR_IBM_ENTITLEMENT_KEY"
```

Login to the docker registry to pull the images

```
echo $IBM_ENTITLEMENT_KEY | docker login -u cp --password-stdin cp.icr.io
```


Step 3. Build Watson Speech-to-Text image

When you cloned the original outline, it pulled a folder called `models` into this workspace. This folder contains the Dockerfiles for the Watson Speech-to-Text images. We will build the models using these Dockerfiles and tag them to later push them to your own registry.

```
cd /home/project/translator-with-voice-and-watsonx/models/stt
docker build ./speech-to-text -t stt-standalone:latest
```

Step 4. Pull and build Watson Text-to-Speech image

Similarly, build the Watson Text-to-Speech image using the Dockerfile in the `models/tts` directory and then tag it to later push it to your own registry.

```
cd /home/project/translator-with-voice-and-watsonx/models/tts
docker build ./text-to-speech -t tts-standalone:latest
```

Step 5. Create a namespace and log in to ICR

You'll need to create a namespace before you can upload your images, and make sure you're targeting the ICR region you want, which right now is `global`.

Choose a name for your namespace, specified as `${NAMESPACE}`, and create the namespace. Currently, it's set to `personal-assistant`, you can choose to rename it to anything you choose.

```
NAMESPACE=personal-assistant
```

```
ibmcloud cr region-set global
ibmcloud cr namespace-add ${NAMESPACE}
ibmcloud cr login
```

Step 6. Push Watson images to your namespace

```
TTS_APPNAME=tts-standalone:latest
STT_APPNAME=stt-standalone:latest
```

```

REGISTRY=icr.io
# Tag and push Text-to-Speech image
docker tag ${TTS_APPNAME}:latest ${REGISTRY}/${NAMESPACE}/${TTS_APPNAME}:latest
docker push ${REGISTRY}/${NAMESPACE}/${TTS_APPNAME}:latest
# Tag and push Speech-to-Text image
docker tag ${STT_APPNAME}:latest ${REGISTRY}/${NAMESPACE}/${STT_APPNAME}:latest
docker push ${REGISTRY}/${NAMESPACE}/${STT_APPNAME}:latest

```

Step 7. Build and push the main app image to own registry

Set a name for your name, by default we will be using watsonx-personal-assistant

```
APP_NAME=watsonx-personal-assistant
```

```

cd /home/project/translator-with-voice-and-watsonx
# Build, Tag and push main App image
docker build . -t ${APP_NAME}:latest
docker tag ${APP_NAME}:latest ${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest
docker push ${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest

```

Step 8. Deploy the images to the Code Engine

Now we will create three applications on Code Engine to deploy our Watson and App images to them.

However, please note these commands assume you have already added a credit card to your IBM Cloud Account and will fail if one has not already been added. Though, you won't be charged until you reach a certain amount of usage. To learn more about IBM Code Engine's free tier, click [here](#).

1: Target a region and a resource group

Choose the region closest to you and/or your target users. Picking a region closer to you or your users makes the browser extension faster. The further the region the longer the request to the model has to travel.

You can choose any region from this list:

Americas

- us-south - Dallas
- br-sao - Sao Paulo
- ca-tor - Toronto
- us-east - Washington DC

Europe

- eu-de - Frankfurt
- eu-gb - London

Asia Pacific

- au-syd - Sydney
- jp-tok - Tokyo

Use the following commands to target Dallas as the region and the Default resource group.

```
REGION=us-south  
RESOURCE_GROUP=Default
```

```
ibmcloud target -r ${REGION} -g ${RESOURCE_GROUP}
```

2: Create and select a new Code Engine project

In this example, a project named `personal-assistant` will be created in the resource group set by the previous command.

```
ibmcloud ce project create --name personal-assistant  
ibmcloud ce project select --name personal-assistant
```

3: Deploy Watson STT and TTS applications

```
ibmcloud ce application create \  
  --name ${STT_APPNAME} \  
  --port 1080 \  
  --min-scale 1 --max-scale 2 \  
  --cpu 2 --memory 8G \  
  --image private.${REGISTRY}/${NAMESPACE}/${STT_APPNAME}:latest \  
  --registry-secret ce-auto-icr-private-${REGION} \  
  --visibility project \  
  --env ACCEPT_LICENSE=true
```

Once the application is ready, it will output a URL in the terminal. Simply go to `yourspeech_to_text` function defined in the `worker.py` file, and then replace the base URL with that.

```
ibmcloud ce application create \  
  --name ${TTS_APPNAME} \  
  --port 1080 \  
  --min-scale 1 --max-scale 2 \  
  --cpu 2 --memory 8G \  
  --image private.${REGISTRY}/${NAMESPACE}/${TTS_APPNAME}:latest \  
  --registry-secret ce-auto-icr-private-${REGION} \  
  --visibility project \  
  --env ACCEPT_LICENSE=true
```

Once the application is ready, it will output a URL in the terminal. Simply go to your `text_to_speech` function defined in the `worker.py` file, and then replace the base URL with that.

4: Deploy your main app

```
ibmcloud ce application create \
  --name ${APP_NAME} \
  --port 8000 \
  --min-scale 1 --max-scale 2 \
  --cpu 1 --memory 4G \
  --image private.${REGISTRY}/${NAMESPACE}/${APP_NAME}:latest \
  --registry-secret ce-auto-icr-private-${REGION} \
  --env ACCEPT_LICENSE=true
```

It may take a few minutes to complete the deployment. If the deployment is successful, you'll get the URL of the application's public endpoint from the command output.

5: Check your deployment

You can check the status, logs and events of the application with the following commands:

```
ibmcloud ce app list
ibmcloud ce app logs --application ${APP_NAME}
ibmcloud ce app events --application ${APP_NAME}
```

Conclusion

Congratulations on completing this guided project on building your own voice assistant with Watsonx! We hope that you have enjoyed learning about assistants and web development, and that you now have the skills and knowledge to improve upon this project.

Throughout this project, you have learned about the various components that make up an assistant, including speech-to-text technology, natural language processing with `mistralai/mistral-large`, text-to-speech technology, and web development using Python, Flask, HTML, CSS, and JavaScript.

Thank you for joining us on this journey to learn about assistants. We encourage you to continue learning and exploring the field of artificial intelligence, and to use your skills to build assistants that are responsible, ethical, and useful to the world. We look forward to seeing what amazing projects you will create in the future!

Next steps

Now that you've built an application using these Speech-to-Text and Text-to-Speech capabilities, if you wish to use IBM Watson Speech Libraries for Embed in your own applications you can use the following links to sign up for free trials.

- [Speech-to-Text](#)
- [Text-to-Speech](#)

The best part about this guided project is you've created a general voice assistant that's capable of so much more through prompt engineering. Prompt engineering is the process of crafting the input to a natural language processing model, such as Watsonx's `mistralai/mistral-large`, to produce a specific type of output. By carefully designing the prompt, you can guide the model to generate responses that are relevant to a particular topic or task.

Here are a few examples of how you can use prompt engineering with `mistralai/mistral-large` to customize the voice assistant for different purposes:

Therapist:

To create an assistant that can provide support and guidance to users who are struggling with mental health issues, you can design prompts that are related to therapeutic conversations. For example, you could start the prompt with "Hello, I'm feeling overwhelmed and stressed today. Can you help me cope with my feelings?" and the assistant could generate a response that provides advice and encouragement.

Mechanic:

To create an assistant that can diagnose and troubleshoot car problems, you can design prompts that ask questions about the symptoms of the car and the possible causes of the problem. For example, you could start the prompt with "My car is making a strange noise when I accelerate. What could be the cause of this?" and the assistant could generate a response that suggests possible solutions based on its knowledge of car repair.

Storyteller:

To create an assistant that can generate original stories on demand, you can design prompts that provide a starting point for the story and let the assistant take it from there. For example, you could start the prompt with "Once upon a time, there was a young princess who lived in a castle. One day, she received a magical gift that changed her life forever. What was the gift and how did it change her life?" and the assistant could generate a unique and engaging story based on that prompt.

Professor:

To create an assistant that can teach users about a specific subject, you can design prompts that provide lesson material and ask questions to test the user's understanding. For example, you could start the prompt with "In this lesson, we will be learning about the properties of matter. What are the three states of matter?" and the assistant could generate a response that provides an explanation and a quiz to test the user's knowledge.

By carefully designing the prompts and using the power of mistralai/mistral-large, you can create an assistant that can perform a wide variety of tasks and provide valuable information to users. The key is to think about the type of output that you want the assistant to generate and design the prompts accordingly. All that's left is creating a beautiful UI/UX around these prompts and you've got yourself a million-dollar business ;)!

Author(s)

Talha Siddiqui

Rohit Arora

Vicky Kuo

Contributor(s)

Kang Wang

© IBM Corporation. All rights reserved.