Enhancing Accessibility for Dyslexic Users within ChatGPT: A User-Centric Approach

Justin Bornais	Nicholas Boisclair	Matthew Connell
110037759	110035144	110023456

University of Windsor

COMP-4990 ~ Project Management - Technique and Tools

Dr. Hossein Fani

January 12, 2024

Project Description:

Accessible Assistant is a web application for ChatGPT that provides accessibility options to make interacting with digital text easier for individuals with dyslexia. It achieves this through the use of text manipulation including increasing or decreasing the font size to reduce the occurrence of letters seeming swapped or “squished” together. Also included is an image generation API that takes the AI responses and generates an image describing the result to avoid issues that come with reading altogether. Finally, it provides a text-to-speech service with controllable text-speed in order to ensure written, visual, and verbal message delivery services are made available for the user.

Project Features Outline

Artificial Intelligence API

OpenAI’s ChatGPT artificial intelligence API was used to generate the response prompts to use within the application. This technology was chosen due to its intuitive API calls as well as its popularity and the frequent updates the Generative model receives.

Text-To-Speech Service

The Coqui TTS Python library is responsible for generating a verbal form of the ChatGPT responses. This service was chosen due to its extensive language support, with up to 16 languages supported simultaneously, as well as its ability to potentially train our own models in the future. Additionally, this technology fits perfectly into our codebase as Python is what hosts the backend; with Coqui TTS being built in Python, it was an intuitive setup process.

Image Generation API

Monster API, a platform for generating text prompts into images using the Stable DIffusion API, was our choice for image generation based on the ChatGPT responses. This was chosen for its ease of use and simplicity. Monster API returns a web URL to the image that is hosted on the CDN, making it simpler to pass this URL directly into an HTML IMG tag on the front end. Additionally, its flexibility for the output through the use of image size parameters as well as the generation of up to 4 images at once made it an obvious choice that will provide room to expand functionality in the future.

User Interface Description

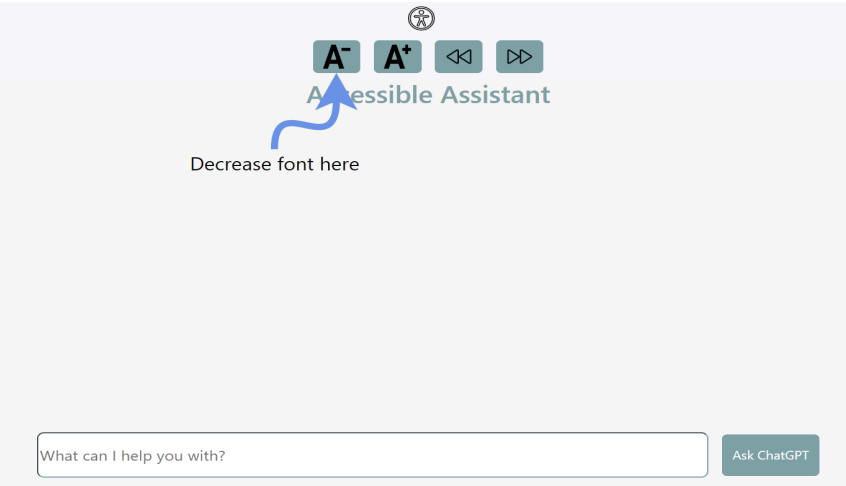
Accessible Assistant's UI provides different options in a hovering menu that control the font-size and voice speed of the text to speech reader. Additionally, on each chat response from ChatGPT, it provides a button to generate an image from the text or convert the text to audio.

Work Breakdown

User Interface Layer

Tutorial Screen

On the main page of the web application being opened for the first time, a tutorial screen showcases the different features. Every time a user clicks, it will explain a new element to ensure smoothless operation with the application. Also, this option only shows up if there are no chats that already exist.

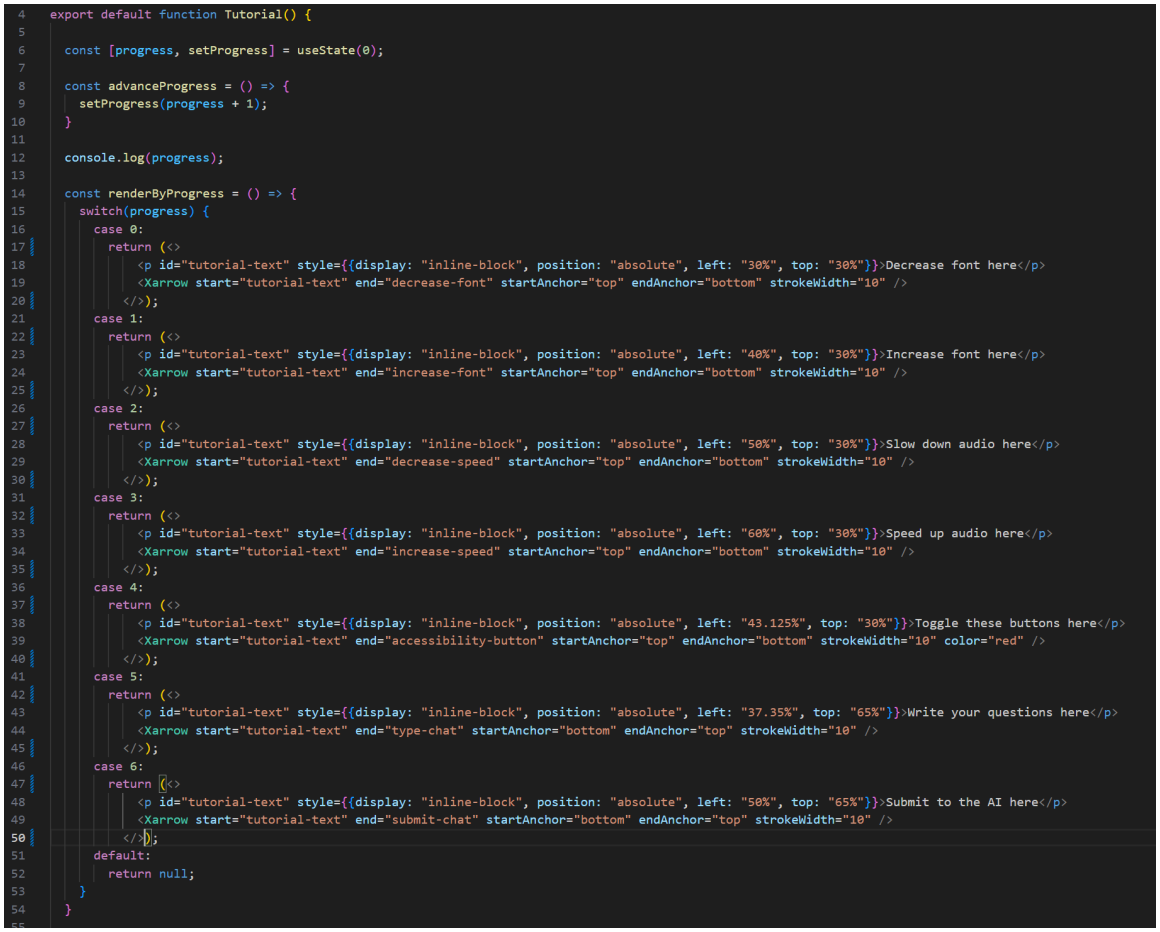


(Figure 0)

This clickable tutorial screen is achieved using a simple fragment calling the function renderByProgress shown in Figure 1, where it uses a switch statement to cycle through the selected button to present tutorial text shown in Figure 2. It then is loaded as a React Fragment and returned to the main page of the application.

```
56    return (
57      <Fragment>
58        {progress < 7 &&
59        <div id="tutorial" onClick={() => advanceProgress()}>
60          {renderByProgress()}
61        </div>
62        }
63      </Fragment>
64    )
65  }
```

(Figure 1)



```
export default function Tutorial() {

  const [progress, setProgress] = useState(0);

  const advanceProgress = () => {
    setProgress(progress + 1);
  }

  console.log(progress);

  const renderByProgress = () => {
    switch(progress) {
      case 0:
        return (<>
            <p id="tutorial-text" style={{display: "inline-block", position: "absolute", left: "30%", top: "30%"}}>Decrease font here</p>
            <Xarrow start="tutorial-text" end="decrease-font" startAnchor="top" endAnchor="bottom" strokeWidth="10" />
          </>);
      case 1:
        return (<>
            <p id="tutorial-text" style={{display: "inline-block", position: "absolute", left: "40%", top: "30%"}}>Increase font here</p>
            <Xarrow start="tutorial-text" end="increase-font" startAnchor="top" endAnchor="bottom" strokeWidth="10" />
          </>);
      case 2:
        return (<>
            <p id="tutorial-text" style={{display: "inline-block", position: "absolute", left: "50%", top: "30%"}}>Slow down audio here</p>
            <Xarrow start="tutorial-text" end="decrease-speed" startAnchor="top" endAnchor="bottom" strokeWidth="10" />
          </>);
      case 3:
        return (<>
            <p id="tutorial-text" style={{display: "inline-block", position: "absolute", left: "60%", top: "30%"}}>Speed up audio here</p>
            <Xarrow start="tutorial-text" end="increase-speed" startAnchor="top" endAnchor="bottom" strokeWidth="10" />
          </>);
      case 4:
        return (<>
            <p id="tutorial-text" style={{display: "inline-block", position: "absolute", left: "43.125%", top: "30%"}}>Toggle these buttons here</p>
            <Xarrow start="tutorial-text" end="accessibility-button" startAnchor="top" endAnchor="bottom" strokeWidth="10" color="red" />
          </>);
      case 5:
        return (<>
            <p id="tutorial-text" style={{display: "inline-block", position: "absolute", left: "37.35%", top: "65%"}}>Write your questions here</p>
            <Xarrow start="tutorial-text" end="type-chat" startAnchor="bottom" endAnchor="top" strokeWidth="10" />
          </>);
      case 6:
        return (<>
            <p id="tutorial-text" style={{display: "inline-block", position: "absolute", left: "50%", top: "65%"}}>Submit to the AI here</p>
            <Xarrow start="tutorial-text" end="submit-chat" startAnchor="bottom" endAnchor="top" strokeWidth="10" />
          </>);
      default:
        return null;
    }
  }
```

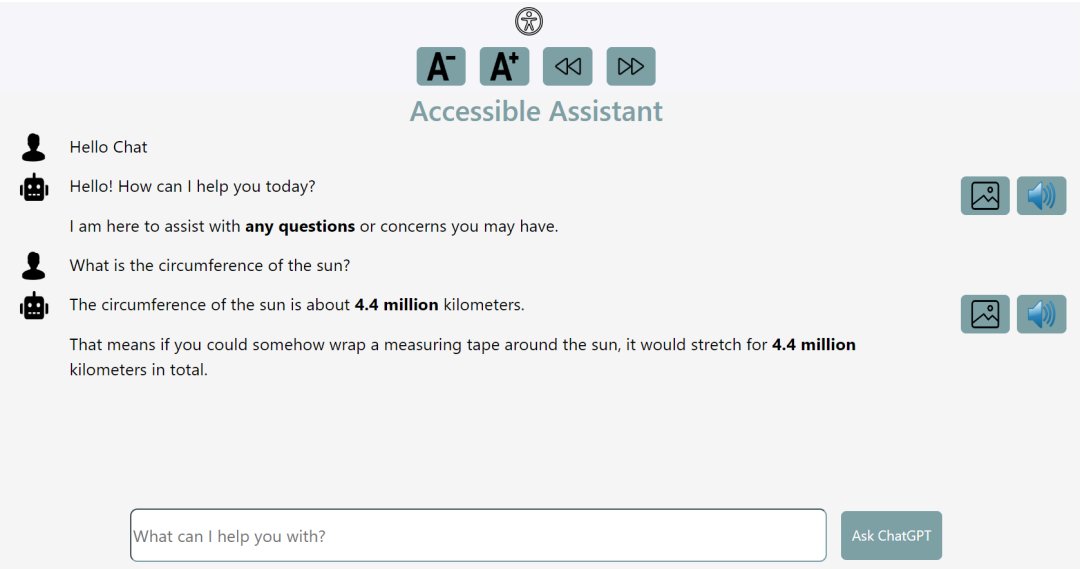
(Figure 2)

To prevent showing the tutorial for revisiting users, the React code will check how many chats currently exist in local storage. If there exists any number of chats, that means the user has already visited this website and does not need the tutorial screen to show. Otherwise, if no chats are visible, then the tutorial will be rendered. This logic is illustrated in Figure 3.

```jsx
import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';
import Chat from './pages/Chat';
import Chats from './pages/Chats';
import Tutorial from './components/Tutorial';
import AccesibilityMenu from './components/AccesibilityMenu';

function App() {

  const chatList = JSON.parse(window.localStorage.getItem('chats')) || [];
  return (
    <Router>
      <AccesibilityMenu />
      <div className='site-content pt-0 p-3'>
        <Routes>
          <Route path='/chats' element={<Chats />} />
          <Route path='/chats/:id' element={<Chat />} />
          <Route exact path='/' element={<Chats />} />
        </Routes>
      </div>
      {chatList.length === 0 && <Tutorial />}
    </Router>
  );
}

export default App;

```

(Figure 3)

Main Chats Page



After entering different prompts, the UI will populate the user's chats and ChatGPT responses in the center of the page as shown above.

The accessibility menu sticks to the top of the screen, and the top icon toggles the menu's visibility. The "**A-**" and "**A+**" correspond to shrinking and growing the application font-size respectively, while the arrow buttons slow down and speed up the text-to-speech responses. These buttons were picked to minimize space while making sure their functionality was easily recognizable.

Figure 4 illustrates the code for changing the font size of the application when clicking the increase or decrease button, while Figure 5 is the code for adjusting the talking speed of the text to speech voice after clicking the arrow buttons.

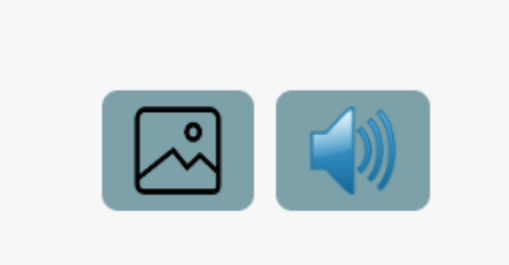
```
const changeFontSize = (action) => {

    // Increase the font size based on action.
    if (action === 'increase') {
        const newSize = fontsize + fontIncrement;
        const newSizeString = `${newSize}rem`;
        root.style.setProperty('--fontsize', newSizeString);
        setFontSize(parseFloat(newSize.toFixed(fontPrecision)));
    }
    else {
        const newSize = fontsize - fontIncrement;
        const newSizeString = `${newSize}rem`;
        root.style.setProperty('--fontsize', newSizeString);
        setFontSize(parseFloat(newSize.toFixed(fontPrecision)));
    }
}
```

(Figure 4)

```
const changeSpeed = (action) => {
    if (action === 'increase') {
        document.querySelectorAll('audio').forEach((audio) => {
            audio.playbackRate = speed + 0.25;
        });
        setSpeed(speed + 0.25);
    }
    else {
        document.querySelectorAll('audio').forEach((audio) => {
            audio.playbackRate = speed - 0.25;
        });
        if (speed >= 0.5) setSpeed(speed - 0.25);
    }
}
```

(Figure 5)

Each AI response will have two buttons corresponding to displaying/generating an image and playing the response audio respectively. The image button will generate an image based on the response ChatGPT provided, while the text to speech button will verbally speak the response ChatGPT provided. These buttons are visible on Figure 6.



(Figure 6)

Figure 7 shows the functionality for generating the chat response. Upon entering a prompt, the addChat function is called, which will send a POST request to the backend server, posting the question being asked. After the fetch request receives a response, it will convert the output into a JSON object paired with the user's prompt, then add it to

the website. This leverages React states to reset the question bar while updating the chat box.

```
const addChat = async () => {
  try {
    const response = await fetch('http://localhost:5000/chats/ask', {
      'method': 'POST',
      headers: {
        'Content-Type':'application/json'
      },
      body: JSON.stringify({ question: question })
    });
    const data = await response.json();
    const newList = chatList.concat({ userName: 'Guest', userMsg: question, AIMsg: data['answer'], id: data['response-id'] });
    window.localStorage.setItem('chats', JSON.stringify(newList)); // Add new prompt to local storage.
    setChatList(JSON.parse(window.localStorage.getItem('chats'))); // Set current state to the new local storage list.
    setQuestion("");
  } catch (error) {
    console.error(error);
  }
}
```

(Figure 7)

The chat instances were created using React's map feature to create a new chat segment for each user prompt and AI response. This also allows each response to be paired with its own independent image generation and text-to-speech buttons. The source code is visible in figure 8. Additionally, the "<Markdown>" tags utilize the "react-markdown" library to render both the user's and AI's responses in markdown. The benefit of this library is it also prevents potential security attacks that deal with custom markdown, such as XSS attacks.

```
<div className="allChats d-flex flex-column" style={{border:"10px"}}>
  {chatList.map((data, index) => {
    return (
      <React.Fragment key={index}>
        <div className="userMessage row">
          <div className="col-2 col-lg-1 d-flex justify-content-end">
            <img className="visualizeImg sticky-top" src="../../images/guest.png" alt="Guest"></img>
          </div>
          <div className="col-10 col-lg-11">
            <Markdown>{data.userMsg}</Markdown>
          </div>
        </div>
        <div className="aiResponse row">
          <div className="col-2 col-lg-1 d-flex justify-content-end">
            <img className="visualizeImg sticky-top" src="../../images/robot.png" alt="AI"></img>
          </div>
          <div className="col-10 col-lg-9">
            <Markdown>{data.AIMsg}</Markdown>
          </div>
          <div className="col-12 col-lg-2 text-center">
            <Button className="m-1 sticky-top" onClick={() => toggleImage(data.id)}>
              <img className="visualizeImg" src="../../images/visualize.png" alt="Toggle"></img>
            </Button>
            <Button className="m-1 sticky-top" onClick={async () => {getAudio(data.id)}}>
              <img className="visualizeImg" src="../../images/audio.png" alt="Play Audio"></img>
              <audio className="chatAudio" src="" controls id={data.id} type="audio/wav" style={{display: "none"}}></audio>
            </Button>
          </div>
        </div>
        <div className="col2 d-flex flex-column align-items-center">
          <img id={`img-${data.id}`} className="visualizedImage" alt="Visualized Result" style={visualizeImage}></img>
        </div>
      </React.Fragment>
    )
  })}
</div>
```

(Figure 8)

Backend Layer

Generating Images

Figure 9 shows the backend function for calling the MonsterAPI and passing in the user's question as the prompt. It then requests a response from the MonsterAPI web service and will return that image url to the frontend to be displayed for the user.

```python
@app.route("/chats/generateImage",methods=['POST'])
def generateImg():
    message = request.json
    print(message['question'])
    #Monster API
    url = "https://api.monsterapi.ai/apis/add-task"

    payload = json.dumps({
    "model": "txt2img",
    "data": {
        "prompt": message['question'],
        "negprompt": "lowres, signs, memes, labels, text, food, text, error, mutant
        "samples": 1,
        "steps": 50,
        "aspect_ratio": "square",
        "guidance_scale": 12.5,
        "seed": 2321
    }
    })
    headers = {
    'x-api-key': '123',
    'Authorization': 'Bearer 456',
    'Content-Type': 'application/json'
    }

    response = requests.request("POST", url, headers=headers, data=payload)
    return response
```

(Figure 9)

Generating Text-To-Speech

Figure 10 shows how the server retrieves the vocals for a ChatGPT response. After the ChatGPT API generates a response, the server immediately creates a new thread and calls the TTS library using the generate_tts function defined in Figure 12, allowing for the server to leverage multithreading to generate the text-to-speech audio. When the user asks for the audio, it calls the get_audio function which will check the status of the audio. If the audio is available, it will return the source audio back to the user. If the audio is still being generated, it will wait for the field to finish before returning. Otherwise, it will call the start_tts_generation function shown in Figure 11 to begin generating the audio.

```python
def get_audio(tts_processes, syn, id):
    field = tts_processes[id]
    print(field)
    while field:
        if field["status"] == "done":
            enc = b64encode(open(f"audio/{id}.wav", "rb").read()).decode('utf-8')
            return str(enc)
        elif field["status"] == "processing":
            time.sleep(0.5)
        else:
            start_tts_generation(tts_processes, syn, id)
            return "nope"

    return "nope"
```

(Figure 10)

```python
def start_tts_generation(tts_processes, syn, id):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(generate_tts(tts_processes, syn, id))
    loop.close()

def first_letters(input):
    input = re.sub(r'[*&^$#@-]', '', input).replace("+", "").replace("\"", "").replace("'", "")
    words = input.split()
    result = ''
    for word in words:
        if word:
            result += word[0]
    return result
```

(Figure 11)

```python
async def generate_tts(tts_processes, syn: Synthesizer, id: str):
    try:
        print(f"Generating audio ID: {id}")
        wav_path = f"audio/{id}.wav"
        abs_wav_path = os.path.abspath(wav_path)
        tts_processes[id]["status"] = "processing"
        text_to_synthesize = re.sub(r'[*&^$#@]', '', tts_processes[id]["answer"])
        text_to_synthesize = text_to_synthesize.replace("\n", ".").replace("\r", "").replace("...", ".").replace("..", ". ").replace(". .", ". ")
        print(f"Synthesizing text: {text_to_synthesize}")
        outputs = syn.tts(text_to_synthesize, split_sentences=True, max_decoder_steps=10000)
        syn.save_wav(outputs, wav_path)

        tts_processes[id]["status"] = "done"
        print(f"Done generating audio ID: {id}")
    except Exception as e:
        tts_processes[id] = {"status": "error"}
        print(f"Error generating audio ID {id}: {str(e)}")
```

(Figure 12)

Receiving ChatGPT responses

Generating ChatGPT responses is simple due to the intuitive API that OpenAI created. The backend receives the message from the frontend and adds the question as “content” for the API, visible in Figure 13. It also adds extra details to the query to ensure ChatGPT generates a short response. After the API creates an answer, the server generates a dynamic ID to begin asynchronously generating the text-to-speech audio. Finally, it returns the response to the front-end where it is shown to the user.

(Figure 13)

Conclusions

Overall, Accessible Assistant is an easy-to-use interface for ChatGPT to improve the experience of people with potential learning challenges, especially dyslexia. We provided options to increase font-size, have a text to speech function with increasing and decreasing reading speed, as well as the ability to generate images for prompts. We also ensured the responses were short and spaced out with key points emboldened using markdown to improve readability.

The system records the user's prompt from the front-end and sends this message to the backend where it is distributed to the targeted service's API. This is then stored to the device's local storage to ensure the progress is saved upon closing and reopening the tab.

Future Plan

Add accessibility options for a larger variety of disabilities

To provide this web application to a larger audience, the Accessible Assistant is planned to be upgraded to have other features related to different disabilities and not limited to only dyslexia. Dyslexia was chosen as our primary focus because of how common it is among the population, as well as other disabilities likely share some of the struggles people with dyslexia deal with. Further features include more web themes for people with visual disabilities such as high contrast, and optional colored text highlighting. This would allow the application to reach a larger audience and ultimately provide an enhanced experience for a wide variety of individuals.

Add an account service

An account service is a future upgrade that would provide some ease of use improvements for users. By implementing an account service, it would allow the chats to be saved on a database along with being stored locally within the device's local storage. This would grant users the ability to access chats from any device and pick up where they left off at any time. Additionally, this would allow them to save their accessibility options to avoid having to readjust all the parameters before using.

Add the ability to generate videos

As of February of 2024, OpenAI released a new technology called Sora that allows the model to generate realistic videos given just a singular prompt. This technology would be beneficial for creating interactive learning material for complex topics or other instances where an in depth explanation is needed that a singular image cannot achieve.

Resources

- Github Repo: *https://github.com/justinbornais/accessible-assistant-499*
- Video Demo of Website:

  https://drive.google.com/file/d/1M0PyD4FE7egqHGkTZ-1aW-IQctDjkhZy/view

References:

Anthony, et al. “6 Surprising Bad Practices That Hurt Dyslexic Users.” *UX Movement*, 11 Dec. 2017, uxmovement.com/content/6-surprising-bad-practices-that-hurt-dyslexic-users/.

“What Is Dyslexia and How Does It Affect the Reading Process.” *Lexia*, 24 Mar. 2023, www.lexialearning.com/blog/what-dyslexia-and-how-does-it-affect-reading-process.

API, Monster. *Monster API*, monsterapi.ai/. Accessed 11 Mar. 2024.

Coqui-Ai. “Coqui-Ai/TTS: 🐸💬 - a Deep Learning Toolkit for Text-to-Speech, Battle-Tested in Research and Production.” *GitHub*, github.com/coqui-ai/TTS. Accessed 11 Mar. 2024.

Openai API, platform.openai.com/docs/introduction. Accessed 11 Mar. 2024.