Designing an Audio-Based Content Moderation System Using Large Language Models

Oleksandr Nediev
Applied Informatics – Software Engineering
Howest University of Applied Sciences
Belgium

Use Case Driven Paper
Course: Professional Networking

2025

Table of Contents

Abstract **3**
Introduction **4**
System Overview and Supporting Literature **5**
Implementation **7**
Moderation pipeline 7
Abstract Definition of a Moderation Step 9
Transcription Step 9
Categorization Step 11
Result preparation step 12
Pipeline Execution via Audio Upload Endpoint 14
Moderation Dashboard 16
Conclusion **19**

Abstract

In this paper, we present a method for building an audio-based content moderation system. Its primary purpose is to guide the reader in creating a pipeline system in the form of an API server that can analyze and moderate audio content. We leverage OpenAI's transcription and moderation APIs to automate the process, making it easier to handle large volumes of content efficiently. As a result, readers will be able to replicate a real-world use case of a highly customizable moderation system, which can serve as a foundation for further development and integration into their own applications.

Additionally, we demonstrate how this moderation system can be applied in the context of a UI dashboard for human content moderators. The dashboard allows moderators to review moderation results and make informed decisions. Its purpose is solely to showcase one possible application of the system, rather than to provide a step-by-step guide for building the dashboard itself.

Introduction

Nowadays, a vast amount of content is generated online, especially on social media platforms where users freely produce audio and visual material. This content can sometimes carry negative connotations and affect the mental well-being of different user groups, which underscores the importance of content moderation. Effective moderation ensures that harmful, misleading, or inappropriate material is identified and managed, thereby protecting users' mental health and fostering a safer digital environment. Hate speech can have profound real-world consequences, including the suppression of marginalized voices, social exclusion, discrimination, and violence against affected groups [1, p. 2].

Proper content moderation can create a sense of comfort and security for all user groups, reducing the likelihood that users will leave a platform in search of safer alternatives.

In this paper, we focus specifically on **audio content moderation**, though many principles also apply to text. Both involve analyzing content to detect potential hate speech, profanity, or harmful intent, whether spoken or written.

Online platforms address moderation challenges using various strategies. One common approach is employing human content moderators to manually analyze and evaluate posted content. However, this approach presents two major challenges:

1. Human moderators are constantly exposed to distressing content, which can negatively affect their mental health [1, p. 3].

2. The volume of content requiring manual moderation is immense, demanding extensive resources.

Audio content moderation poses additional challenges. Unlike text, audio conveys harmful content through tone, context, and implicit cues, making it harder to detect without advanced tools. Unmoderated audio can amplify hate speech, harassment, and misinformation, disproportionately affecting vulnerable groups. Effective moderation is crucial to maintain user trust, comply with regulations, and promote healthier online communities.

To address these challenges, platforms increasingly leverage **large language models (LLMs)** to automate at least part of the moderation process. Recent studies have shown that LLMs achieve state-of-the-art performance in natural language tasks and demonstrate strong contextual understanding, making them effective for detecting verbal hate speech and explaining rule violations in online communities [2, p.3].

System Overview and Supporting Literature

There are several approaches to using LLMs in a content moderation loop for audio data. In this section, we describe the crucial steps and decision-making involved in designing a moderation pipeline, providing reasoning for the choices made during its construction.

First, we design our system as a **pipeline**. This architecture ensures high customizability, allowing different steps to be swapped, omitted, or added without disrupting the overall flow of the moderation system. A pipeline-based design also provides **modularity**, enabling each component—such as speech-to-text, classification, or reasoning—to be independently improved or replaced as new tools and models emerge. This flexibility makes the system practical and straightforward for readers to implement and tailor to specific use cases.

The next step in our moderation service is defining **how the audio content will actually be moderated**. The most widely-used approach is to transcribe audio using speech-to-text LLMs. Transcription converts spoken content into text, which can then be analyzed and moderated using standard text-based methods. While this approach is simple and effective, it has limitations—primarily its inability to account for the **emotional tone or prosody** of speech.

To address this limitation, **Mel-spectrograms** can be used. A Mel-spectrogram is a heat-map representation of audio that can reveal specific patterns associated with hate speech. Recent work by Roblox highlights the effectiveness of combining proprietary text filters with audio feature analysis to detect hate speech [2, p.4]. Their results indicate that incorporating audio features can improve detection accuracy and reduce false positives.

However, GPT models alone already demonstrate strong performance in detecting hate speech. While combining text and audio features is promising, in our system we focus solely on using GPT for moderation. Nonetheless, it is important to note the potential of hybrid methods, and we leave the exploration of combined approaches to readers interested in further research.

The next crucial step is to select the **LLM for moderation**. While there are many models available on the market, each is trained on different datasets and may exhibit under- or over-moderation tendencies. It is important for our system to choose a model that minimizes bias, particularly towards vulnerable groups. In tests of the five most widely used commercial moderation APIs, OpenAI's moderation tool demonstrated the most consistent performance and least biased behavior, followed closely by the Perspective API [1, Table 3, p. 9]. For this system, we will use OpenAI's moderation model. An additional advantage is that it is free to use, which makes it practical for implementation.

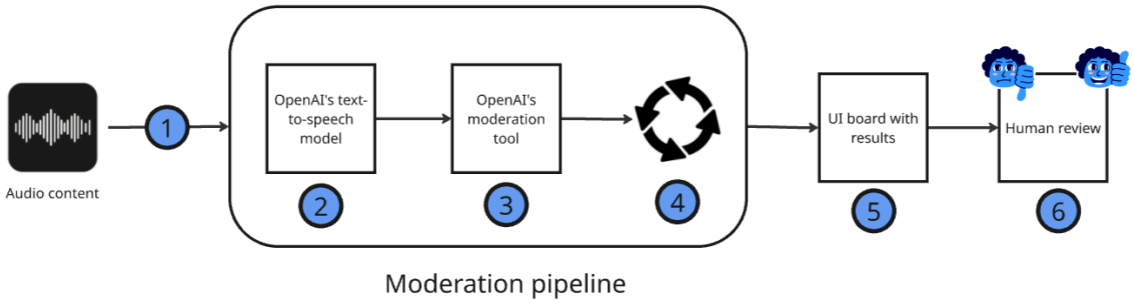
uch moderation tools return a **probability of violations** across multiple categories, such as profanity, hate speech, and other forms of harmful content. Once these probabilities are obtained, the results must be processed, filtered, and presented in a clear and accessible manner.

The next question is how to handle flagged content. Strategies for managing such content depend on a platform's policies and goals. Common actions include restricting visibility,

deleting content, sending it for review by a human moderator (human-in-the-loop), or using additional AI agents to confirm results. Other practical strategies include shadow banning, notifying or warning the user, temporarily limiting account activity, rate-limiting content reach, automatically labeling content as sensitive, or escalating issues for legal or regulatory compliance. The choice of strategy typically balances **safety, user experience, and regulatory requirements**.

In our specific use case, we implement a **moderation dashboard** that flags content based on the model's results and forwards it for human review. This approach reduces moderator workload and automates routine monitoring, while ensuring that only potentially harmful content requires manual inspection.

Having defined the key components of the system and established its blueprint, we can now proceed to its implementation, as illustrated in **Figure 1**.



1. Provide audio content
2. Transcribe the audio
3. Categorize the input text with a moderation tool
4. Process the results
5. Present them to the human moderator
6. Moderator flags the content and chooses the action (human-in-the-loop)

Figure 1. Overview of the audio-based content moderation system.

Implementation

For the implementation in this UCDP, we build the previously described pipeline for audio moderation using TypeScript. An Express.js server is set up with a single endpoint to execute the moderation pipeline on a provided audio file.

For this specific use case, we also develop a simple moderation dashboard in React to demonstrate how the pipeline can be used in practice. Its sole purpose is to display moderation results and allow human moderators to make decisions based on the outputs of the LLM. We provide an overview of the dashboard's interface and a few technical insights, but it is not the objective of this UCDP to detail the process of building the dashboard.

The full code for the moderation pipeline and dashboard is available on GitHub:

- Moderation Pipeline Repository
- Moderation Dashboard Repository

Moderation pipeline

We begin by implementing the **`ModerationPipeline`** class. This class stores all provided moderation steps in sequence and executes them one by one as needed.

Upon creation of a pipeline, an **audio context object** is initialized. This object persists throughout the lifecycle of the pipeline and is passed through each moderation step. It is constructed from the audio file provided by the user, allowing the system to process content sequentially while retaining the intermediate data of each step.

The structure of the audio context object is as follows:

```
type AudioContextBundle = {
    id: string;
    name: string;
    uploadedFile: UploadedFile
    transcriptions?: Transcriptions;
    moderation?: ModerationContextBundle;
};
```

The pipeline class has two functions that let it work properly:

1. **use(step: ModerationStep)** - it takes a moderation step object as an argument and saves it in an array property of the pipeline class.

2. **run()** - iterates over each step, runs it and updates the returned by each step context.

```
class ModerationPipeline{
    private steps: ModerationStep[] = []
    private ctx: AudioContextBundle;

    constructor(audioFile: UploadedFile){
        this.ctx = {
                id: uuid(),
                name: `${uuid()}-${audioFile.originalName}`,
                uploadedFile: {
                    buffer: audioFile.buffer,
                    originalName: audioFile.originalName,
                    mimeType: audioFile.mimeType,
                }
        }
    }

    use(step: ModerationStep){
        this.steps.push(step);
        return this;
    }

    async run(): Promise<AudioContextBundle>{
        let resultCtx: AudioContextBundle = this.ctx;
        for(const step of this.steps){
            resultCtx = await step.execute(resultCtx);
        }
        return resultCtx;
    }
}
```

An **`UploadedFile`** is a type representing an audio file provided for moderation. It mirrors the properties of the original file and is used primarily for development convenience, ensuring consistent handling of audio data within the system.

We also use the **`uuid`** library to generate unique identifiers for each uploaded audio file and its corresponding moderation context. This ensures that every file and its associated data can be reliably tracked throughout the moderation pipeline.

Abstract Definition of a Moderation Step

Moderation steps are executed sequentially by the pipeline. Each step is implemented as a class that adheres to a common **`ModerationStep`** interface:

```
export interface ModerationStep{
    stepName: string;
    description: string;
    tool?: AI_Tool;
    execute: (ctx: AudioContextBundle) => Promise<AudioContextBundle>;
}
```

The `stepName` and `description` properties serve purely as metadata. Optionally, a step can be provided with an **`AI_Tool`** object, which represents the LLM or AI tool used for that step:

```
export type AI_Tool = {
    model: string;
    api_endpoint: string;
    api_key: string;
    prompt?: string;
}
```

This design provides flexibility, allowing different tools to be swapped in or out easily, as long as they follow a similar structure.

The core of each step is the **`execute()`** function, which takes an **`AudioContextBundle`** as input and returns it as output. This ensures smooth data flow between steps within the pipeline.

In our implementation, three classes follow this interface: **Transcription**, **Categorization**, and **Result Preparation**. Each of these steps will be discussed in detail in the following sections.

Transcription Step

The **Transcription** step implements the **`ModerationStep`** interface. This class is initialized via a constructor that takes the step's name, description, and the AI tool used in this step.

For transcription, we use OpenAI's **Speech-to-Text API**, which allows us to convert the audio content provided by the user into text. In this implementation, to simplify processing, we use the **`/translations`** endpoint. This endpoint automatically transcribes audio in any language and translates it into English. The corresponding tool is defined in our code as follows:

```
export const OPENAI_TRANSCRIPTION_API: AI_Tool = {
    model: "whisper-1",
    api_endpoint: "https://api.openai.com/v1/audio/translations",
    get api_key() {
        return process.env.OPENAI_API_KEY?.trim() ?? "";
    }
}
```

When this step is executed, the pipeline calls the transcription API using the provided API key. The response contains the transcription of the audio, which is then added to the **AudioContextBundle**. The updated context bundle is returned to the pipeline for use in subsequent moderation steps.

```
async execute(audioContextBundle: AudioContextBundle):
Promise<AudioContextBundle> {
        const text = await
this.transcribe(audioContextBundle.uploadedFile)
        audioContextBundle = {
            ...audioContextBundle,
            transcriptions: {"en" : text},
        }
        return audioContextBundle;
    }

    async transcribe(audioFile: UploadedFile): Promise<string> {
        try {
            const formData = new FormData();
            formData.append("model", "whisper-1");
            formData.append("file",
                new File(
                    [this.bufferToArrayBuffer(audioFile.buffer)],
                    audioFile.originalName,
                    { type: audioFile.mimeType }
                )
            );
            const response = await axios.post(
                this.tool.api_endpoint,
                formData,
                {
                    headers: {
                        Authorization: `Bearer ${this.tool.api_key}`,
                    },
                }
            );
            return response.data.text;
        } catch (e: any) {
```

```
            throw new Error(`Error while transcribing and translating:
${e.message || e}`);
        }
    }
```

Categorization Step

The **Categorization** step is responsible for the actual moderation of the transcribed text. In this step, we call OpenAI's **Moderation API**, which takes the text as input and returns a result object containing several fields. For our purposes, the most important fields are:

1. **`flagged`** – indicates whether the text violates any rules and has been flagged.
2. **`categories`** – key-value pairs indicating which categories of policy violations apply.
3. **`category_scores`** – key-value pairs where each key represents a category and the corresponding value is the probability of violation for that category (ranging from 0 to 1).

The **Categorization** step follows the same structure as the previous step. We make a call to the Moderation API using the AI tool defined as follows:

```
export const OPENAI_MODERATION_API: AI_Tool = {
    model: "omni-moderation-latest",
    api_endpoint: "https://api.openai.com/v1/moderations",
    get api_key() {
        return process.env.OPENAI_API_KEY?.trim() ?? "";
    }
}
```

When this step is executed, the transcription stored in the **AudioContextBundle** is sent to the Moderation API via an API request. The response is then assigned to the **moderation bundle** within the audio context. The moderation bundle contains the full result returned by the API, including a **`violations`** object, which will play a critical role in subsequent steps of the pipeline.

```
export type ModerationContextBundle = {
    result: ModerationResult;
    violations: Partial<Record<ViolationType, Array<[string, number]>>>
}

export type ModerationResult = {
  flagged: boolean;
  categories: {
    [category: string]: boolean;
  };
  category_scores: {
        [category: string]: number;
  };
};
```

We save the moderation results in the context and return the context for the last step of our pipeline. Now the execute function will look like this:

```
async execute(audioContext: AudioContextBundle):
Promise<AudioContextBundle> {
        try{
            const transcription = audioContext.transcriptions?.["en"] ??
"";
            if(transcription === ""){
                throw new Error("Transcription is empty");
            }
            const moderationCategories: ModerationResult = await
this.moderate(transcription);

            audioContext.moderation = {
                result: moderationCategories,
                violations: {}
            };
            return audioContext;
        } catch (e: any) {
            throw new Error(`Error while getting moderation: ${e.message
|| e}`);
        }
    };
```

Result preparation step

To separate responsibilities within the pipeline, we introduce the **PrepareResultStep**. In this step, all moderation results are processed and organized into a format that is most

convenient for subsequent use. Essentially, this step filters and sorts the results to provide a structured output tailored to the needs of our system.

No AI tool is associated with this step, as it does not involve any API calls or external processing.

For this implementation, we define a **`violations`** object to structure the moderation results. The object uses an enum of violation severity levels as keys, and each key maps to an array containing the violated categories along with their corresponding result values. The types and enum used for this object are as follows:

```
export type ModerationContextBundle = {
    result: ModerationResult;
    violations: Partial<Record<ViolationType, Array<[string, number]>>>
}

export enum ViolationType {
    HIGH_RISK = "high_risk",
    MEDIUM_RISK = "medium_risk",
    LOW_RISK = "low_risk",
}
```

And this is how the actual result looks like:

```
"violations": {
        "high_risk": [
            [
                "harassment",
                0.9788440980829426
            ]
        ],
        "medium_risk": [],
        "low_risk": []
    }
```

In this step, we process the results obtained from the Moderation API and apply additional filtering. If the text is not flagged, this step is skipped, as the content does not violate any rules. Although OpenAI's default threshold for flagging text is 0.8, we apply our own filtering to assess the **severity** of violations for each category. This approach provides human moderators with deeper insights into the moderation results, enabling them to evaluate the audio more effectively.

The **`execute()`** function of the **PrepareResultStep** is implemented as follows:

```
async execute(bundle: AudioContextBundle): Promise<AudioContextBundle> {
        const violations: Partial<Record<ViolationType, Array<[string,
number]>>> = {
            [ViolationType.HIGH_RISK] : [],
            [ViolationType.MEDIUM_RISK] : [],
            [ViolationType.LOW_RISK] : [],
        };
        if(bundle.moderation?.result === undefined){
            throw new Error("No moderation result found to process
it.");
        }

        const moderationResult: ModerationResult =
bundle.moderation.result;

        if(!moderationResult.flagged){
            return bundle;
        }

        Object.entries(moderationResult.category_scores).forEach(([key,
value]) => {
            if (value >= 0.8) {
                violations[ViolationType.HIGH_RISK]!.push([key, value]);
            } else if (value >= 0.5 && value < 0.8) {
                violations[ViolationType.MEDIUM_RISK]!.push([key,
value]);
            } else if (value >= 0.2 && value < 0.5) {
                violations[ViolationType.LOW_RISK]!.push([key, value]);
            }
        });

        bundle.moderation = {
            ...bundle.moderation,
            violations,
        };

        return bundle;

    }
```

Pipeline Execution via Audio Upload Endpoint

This final part of the moderation pipeline brings all components together and enables the system to function end-to-end.

For the `/uploadAudio` endpoint, we use the **Multer** library to handle incoming audio files in HTTP requests. The uploaded file is then extracted from the request and converted into our `UploadedFile` type, allowing it to be processed seamlessly by the moderation pipeline.

```
app.post(
  "/uploadAudio",
  upload.single("audioFile"),
  async (req: Request, res: Response) => {
    if (!req.file) {
      return res.status(400).send("No file uploaded.");
    }

    const file: UploadedFile = {
      originalName: req.file.originalname,
      buffer: req.file.buffer,
      mimeType: req.file.mimetype,
    };
```

Next, we initialize the **moderation pipeline** and instantiate its individual steps. The steps are then chained sequentially, allowing the pipeline to execute each function in order. Once the pipeline runs, it returns the **finalized AudioContextBundle**, containing all intermediate and processed results from each step.

```
const pipeline = new ModerationPipeline(file);

      const transcription = new TranscriptionStep(
        "Transcription Step",
        "This is a step to request transcription of audio translating in
English language from OpenAI Whisper API.",
        OPENAI_TRANSCRIPTION_API
      );

      const moderation = new CategorisationStep(
        "Categorisation Step",
        "This step is used to moderate transcribed text from audio and
classify the content using OpenaAI API.",
        OPENAI_MODERATION_API
      );

      const preparation: PrepareResultsStep = new PrepareResultsStep(
        "Preparation Result Step",
        "This step is needed to sort the moderation results
appropriately"
      )

      const resultCtx = await pipeline
```

```
        .use(transcription)
        .use(moderation)
        .use(preparation)
        .run();
```

After this we create a data transfer object (DTO) to shape our context into a suitable format to send as a response.

```
const outputDTO: ModerationOutputDTO = {
        id: resultCtx.id,
        name: resultCtx.name,
        originalName: resultCtx.uploadedFile.originalName,
        transcription: resultCtx.transcriptions ?? {},
        flagged: resultCtx.moderation?.result.flagged ?? false,
        violations: resultCtx.moderation?.violations ?? {},
 };

 res.json(outputDTO);
```

Like this we created a full working pipeline to moderate the provided audio content.

Moderation Dashboard

The **Moderation Dashboard** is implemented in pure **React**. It is a simple web application designed to test the moderation pipeline and display its results. We do not focus on the implementation details, as the primary objective of this UCDP is the pipeline itself.

The dashboard provides three main functionalities:

1. Display moderated audio entries in a table.
2. Submit audio files to the server for moderation.
3. Allow human moderators to take actions based on the moderation results.

Figure 2. Moderation dashboard

For the user interface, we used the **shadcn** library to quickly build visually appealing components.

Regarding the front-end logic, the dashboard sends an **Axios** request to the server with the uploaded audio file. The server responds with a `ModerationResponse` object, a type specifically designed to encapsulate moderation data. This object includes an additional field to record the decision made by the human moderator.

```
export type ModerationResponse = {
    id: string;
    name: string;
    originalName: string;
    transcription: Transcriptions;
    flagged: boolean;
    violations: Partial<Record<ViolationType, Array<[string, number]>>>;
    reviewStatus: "Pending" | "Blocked" | "Approved";
}
```

As a result, the human moderator can view a **modal** displaying the transcribed content, the flagged categories, and the corresponding probabilities indicating the LLM's confidence that each category has been violated.

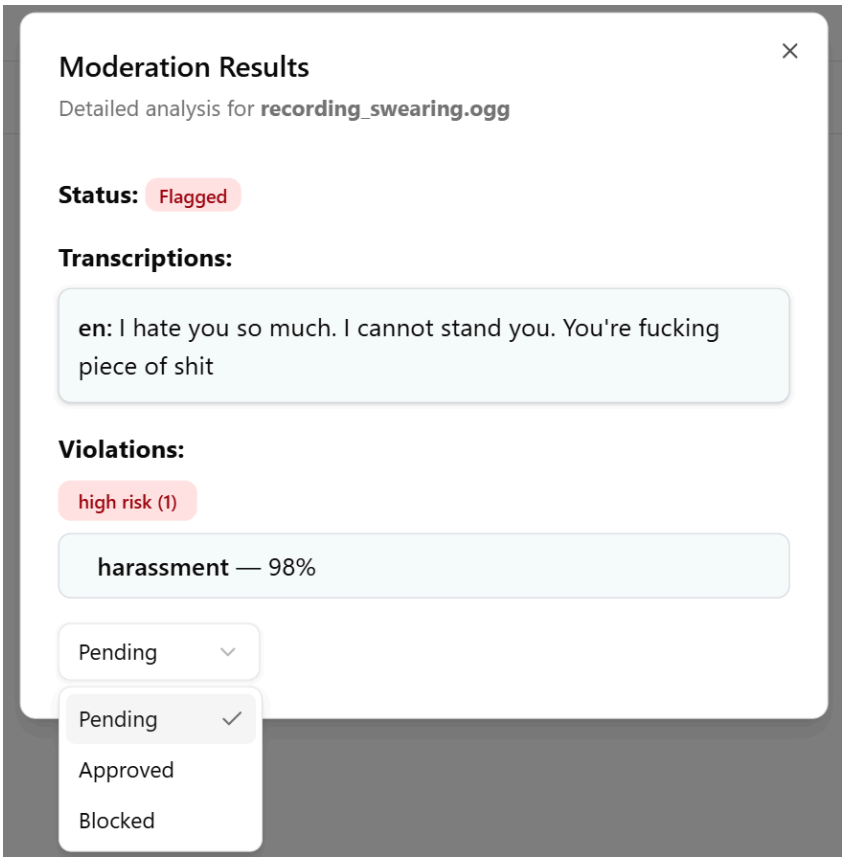


Figure 3. Moderation result modal

This represents just one possible way to utilize the moderation results within a system. For those interested in the full implementation details of the dashboard, the corresponding GitHub repository can be consulted.

Conclusion

In today's digital world, audio content has become increasingly popular, and the sheer volume makes manual moderation by humans impractical. This UCDP explores different approaches to audio content moderation and demonstrates a flexible, pipeline-based solution using LLMs. The proposed system provides readers with a concrete foundation for building an audio moderation system from scratch, while maintaining modularity and extensibility.

The implementation emphasizes adaptability, allowing readers to integrate alternative tools or expand the pipeline with additional steps to suit specific use cases. For instance, extra GPT moderation steps with customized prompts or emotion recognition using heat maps and Mel-spectrograms could enhance the precision of moderation results. Such extensions can improve automated decision-making and provide a more robust and intelligent moderation system.

Moreover, this architecture can be deployed using cloud services to automate moderation at scale. Cloud functions can trigger the pipeline on audio uploads, while message brokers can orchestrate processing, ensuring a scalable, reliable, and near real-time moderation workflow.

Overall, this UCDP provides not only a functional proof-of-concept but also a roadmap for future research and development in automated audio content moderation. By combining LLMs, modular pipeline design, and cloud-based infrastructure, the system can evolve to meet the growing demands of online platforms and contribute to safer digital environments.

References

[1] D. Hartmann, A. Oueslati, D. Staufer, L. Pohlmann, S. Munzert, and H. Heuer, “Lost in Moderation: How Commercial Content Moderation APIs Over- and Under-Moderate Group-Targeted Hate Speech and Linguistic Variations,” TU Berlin & Weizenbaum Institute for the Networked Society, Berlin, Germany, 2025.

[2] Y. Xu, Q. Hou, H. Wan, and M. Prpa, “Safe Guard: An LLM-agent for real-time voice-based hate speech detection in social virtual reality,” Northeastern University, Khoury College of CS, Vancouver, Canada, 2024.