📄 REMORA — VR Simulation Project Documentation

🧾 Project Charter

Remora — Virtual Reality Simulation: AI-Powered Social Conversation Trainer

Date: June 2025
Version: 1.0
Project Name: Remora
Author: Namir Salaheddine
Platform: Unity for Meta Quest 2 / PC VR
Project Type: Virtual Reality Simulation

Project Overview

Remora is a virtual reality simulation designed to offer users an immersive environment in which they can engage in realistic, low-pressure social interactions with a shopkeeper NPC. The shopkeeper's responses are generated using OpenAI's ChatGPT, and user speech is transcribed via OpenAI Whisper. The goal is to provide a safe and repeatable social training tool, particularly for neurodiverse individuals, such as those on the autism spectrum, who may experience challenges in real-world social situations.

Project Purpose

The purpose of this project is to explore the integration of conversational AI into a VR context to simulate everyday human interaction in a structured and non-threatening setting. The intention is to support the development of social communication skills through interaction with a friendly and context-aware virtual agent in a controlled virtual space.

Project Objectives

Design and implement a small retail store environment in virtual reality.

Enable user interaction with an NPC through both voice and text input.

Use OpenAI ChatGPT to dynamically generate context-sensitive responses.

Use OpenAI Text-to-Speech (TTS) to vocalize the NPC's replies.

Display NPC responses in a speech bubble above the character.

Incorporate button-based prompts as an alternate interaction method.

Record user voice input via microphone and transcribe it using Whisper API.

Ensure stable performance on Meta Quest 2 hardware.

Structure the application to allow for future extension to additional scenarios or locations.

Project Scope

In Scope:

3D store layout compatible with VR interaction and locomotion

NPC shopkeeper prefab with animation, audio playback, and UI integration

UI components for initiating conversation (e.g., “Ask a question”)

API communication with OpenAI ChatGPT, Whisper, and TTS services

Voice capture and audio preprocessing pipeline for speech-to-text

Feedback UI elements to indicate when the system is listening or replying

Out of Scope (Current Version):

Facial animation or lip sync beyond audio playback

Long-term conversational memory or emotional context retention

Multiplayer or multi-NPC interactions

Voice cloning or advanced personalization of the NPC

Key Deliverables

Unity project with VR-ready simulation scene

NPC character with full voice interaction loop

Scripted microphone recording system and audio file generation

Whisper API integration for voice-to-text

ChatGPT integration for conversational generation

TTS integration for synthesized speech playback

Setup and configuration documentation for deployment and usage

Milestones

Milestone	**Status**
VR Store Environment Setup	Complete ✅
ChatGPT Dialogue Integration	Complete ✅
TTS Voice Synthesis	Complete ✅
Dialogue Display (Text Bubble)	Complete ✅
Voice Input (Mic + Whisper)	Complete ✅
Documentation	Complete ✅

Stakeholders

Project Author: Namir Salaheddine

Intended Users: Neurodiverse individuals practicing conversational skills

ALGOSUP: School who supervise the project

Assumptions and Constraints

Assumptions:

The target device has a working microphone and stable internet access.

OpenAI API credentials are available and within usage limits.

Users are able to engage with VR input methods (controllers or hand tracking).

Constraints:

Performance limitations on standalone VR hardware (Meta Quest 2).

Latency introduced by API response times.

User privacy considerations related to audio input and AI processing.

Success Criteria

The system allows users to initiate and complete basic conversations with the NPC using voice.

The dialogue generated is coherent, friendly, and contextually relevant.

Text and speech feedback are synchronized and clear.
The system performs reliably across multiple conversation turns without critical failures.

🧩 Functional Specification Document

Project Name: Remora

Version: 1.0

Author: Namir Salaheddine

Date: June 2025

Platform: Unity for Meta Quest 2

Introduction

1.1 Purpose

This document outlines the functional specifications of the Remora VR project — a simulation designed to provide a safe, interactive environment for users, particularly neurodiverse individuals, to practice social interactions using a voice-enabled, AI-powered NPC.

1.2 Target Audience

This simulation is designed for individual users who require support in practicing everyday social communication, and for educators, therapists, or researchers evaluating virtual tools for behavioral training.

1.3 Scope

This version of Remora is focused on simulating a single retail store interaction involving one shopkeeper NPC and a user. It supports both text-based and voice-based interaction using natural language processing and speech synthesis via OpenAI services.

System Overview

2.1 Architecture Summary

Engine: Unity (VR-capable)

Hardware Target: Meta Quest 2

Language Processing: OpenAI ChatGPT API

Text-to-Speech: OpenAI TTS API

Speech-to-Text: OpenAI Whisper API

Input: VR controller, or microphone

Output: Audio speech, 3D UI text bubble

Functional Requirements

3.1 Environment Setup

Name: VR Store Scene
Description: A simple, stylized virtual store containing product shelves, signage, and walkable space with locomotion or teleportation support.
Status: Implemented ✅

3.2 Shopkeeper NPC

Name: AI-Powered NPC

Description: A stationary humanoid character representing a shopkeeper. NPC responds to user questions using voice and text bubbles.
Status: Implemented ✅

3.3 Chat-Based Dialogue System

Name: Button-Based Prompt System
Description: Users can click predefined options like “Ask for help” to trigger a chat prompt to ChatGPT.
Status: Implemented ✅

3.4 Voice Interaction System

Name: Microphone Input + Transcription
Description: The system records voice input from the user, converts it to a .wav file, and sends it to OpenAI Whisper for transcription.
Status: Implemented ✅

Name: Whisper Integration
Description: The audio file is sent via HTTP to OpenAI’s Whisper API. The returned text is forwarded to the chat system.
Status: Implemented ✅

3.5 Conversational AI

Name: ChatGPT Integration
Description: The input (button text or Whisper transcription) is sent to OpenAI’s ChatGPT API with pre-defined prompt context (e.g. “You are a helpful shopkeeper”).
Status: Implemented ✅

3.6 Response Rendering

Name: NPC TTS Voice
Description: ChatGPT’s response is sent to OpenAI TTS API and returned as an mp3. The result is played aloud using Unity’s AudioSource.
Status: Implemented ✅

Name: Text Bubble Display
Description: The spoken message is also rendered above the NPC in a floating UI bubble.

Status: Implemented ✅

3.7 Feedback and State UI

Name: Listening Indicator
Description: A UI element (e.g. an icon or animation) indicates when the system is recording or waiting for a response.
Status: Implemented ✅

Non-Functional Requirements

4.1 Performance

Whisper, ChatGPT, and TTS round-trip latency should remain under ~5 seconds

4.2 Accessibility

Button interface remains available as fallback to voice

UI text uses large, readable fonts and simple language

4.3 Network

Requires continuous internet access for API communication

4.4 Data Privacy

No user voice/audio is stored locally unless explicitly allowed

API keys are securely stored and not embedded in public builds

4.5 Compatibility

Must run on Unity 2021.3 LTS or later

User Flows

5.1 Button-Based Conversation

User enters store

User selects a prompt from UI

System sends prompt to ChatGPT

NPC speaks response via TTS

NPC shows text bubble of response

5.2 Voice-Based Conversation

User speaks (via mic hotkey or auto-trigger)

Audio is recorded and saved as .wav

File is sent to Whisper API for transcription

Transcribed text is sent to ChatGPT

Response is voiced via TTS and shown as text

Current Limitations

No facial animation or lip sync

No persistent memory across sessions

No error handling for poor or missing mic input

Only one NPC and one interaction context supported

🛠️ Technical Specification Document

Project name: Remora

Version: 1.0

Author: Namir Salaheddine

Date: June 2025

Platform: Unity (Meta Quest 2)

Introduction

This document outlines the technical architecture and implementation details of Remora, a VR application that enables voice-based conversations with an AI-powered NPC using Unity and OpenAI APIs. It provides guidance for developers on how the system is structured, how its components interact, and how to extend or maintain the project.

Technology Stack

Engine: Unity 6000.0.43f1

Target Platform: Meta Quest 2

Language: C#

Input: Unity Microphone API, XR Toolkit

Audio: Unity AudioSource, OpenAI TTS (MP3)

AI APIs: OpenAI ChatGPT (text), Whisper (audio transcription), TTS (speech synthesis)

Networking: UnityWebRequest with HTTP POST (RESTful JSON/multipart)

File Format: WAV (for Whisper), MP3 (for TTS)

UI Framework: Unity UI Toolkit / Canvas-based UI

System Architecture

Subsystems:

UI Layer: Buttons, text bubble, feedback icons

Input Layer: Mic recording, button press detection

Networking Layer: API requests and response parsing

AI Layer: ChatGPT (text), Whisper (speech-to-text), TTS (text-to-speech)

NPC Layer: Audio playback, dialogue display, animation (future)

Data Flow:

User speaks → Audio recorded → Saved as WAV → Sent to Whisper → Transcript → Sent to ChatGPT → Response → Sent to TTS → MP3 played & displayed as text

🔧Modules and Components

This section describes the core C# scripts responsible for orchestrating the AI-driven NPC interaction pipeline. These components are loosely coupled and interact through clearly defined function calls and Unity MonoBehaviour event handling.

4.1 DialogueManager.cs

📌 Purpose:

Acts as the central orchestrator for conversation logic. It bridges user inputs (text or voice) with AI-generated replies and handles the routing of messages between UI and audio subsystems.

🛠️ Key Responsibilities:

- Receives user inputs (from button clicks or Whisper transcription)

- Sends input to the OpenAIChat module for processing

- Receives generated responses from ChatGPT

- Triggers OpenAITTS to vocalize the reply

- Updates the UI (e.g., text bubble) to show what the NPC is saying

🔁 Typical flow:

```
public void SendUserSpeech(string transcript)
{
    chat.SendPrompt(transcript); // Calls OpenAIChat
}
```

💬 Output:

- Calls OpenAITTS.Speak(response)

- Calls UIManager.ShowTextBubble(response)

🔗 Dependencies:

- OpenAIChat

- OpenAITTS

- UI (e.g., TextMeshPro or speech bubble handler)

4.2 MicRecorder.cs

📌 Purpose:

Manages audio input from the user's microphone, handles WAV encoding, and sends voice input to OpenAI Whisper for transcription.

🛠️ Key Responsibilities:

- Starts and stops microphone recording

- Converts AudioClip into a WAV file using a custom WAV encoder

- Sends WAV file to OpenAI Whisper API for transcription

- Forwards the recognized text to DialogueManager

🎙️ Code snapshot:

```
public void StopRecordingAndSend()
{
    Microphone.End(null);
    SaveWav(filePath, recordedClip);
    StartCoroutine(SendToWhisper(filePath));
}
```

📥 Output:

- Calls DialogueManager.SendUserSpeech(transcribedText)

🔗 Dependencies:

- Microphone API

- UnityWebRequest (POST to Whisper endpoint)

- DialogueManager

4.3 OpenAIChat.cs

📌 Purpose:

Manages communication with OpenAI's ChatGPT API to generate responses from the NPC based on user input.

🛠️ Key Responsibilities:

- Constructs a valid chat/completions request (system + user messages)

- Sends the request to OpenAI using UnityWebRequest

- Parses the assistant’s reply from the JSON response

- Sends the result back to DialogueManager

💬 Request example:

```
{
  "model": "gpt-4o",
  "messages": [
    { "role": "system", "content": "You are a helpful shopkeeper..." },
    { "role": "user", "content": "Where can I find bread?" }
  ]
}
```

🧠 Code logic:

```
public void SendPrompt(string userInput)
{
    // Build JSON and send POST
    // On success, call dialogueManager.OnAIReply(response);
}
```

📤 Output:

- Forwards raw reply text to TTS + UI

🔗 Dependencies:

- UnityWebRequest

- DialogueManager

4.4 OpenAITTS.cs

📌 Purpose:

Handles Text-to-Speech functionality using OpenAI’s /v1/audio/speech endpoint. Converts AI responses into audio and plays them back via AudioSource.

🛠️ Key Responsibilities:

- Prepares a JSON payload with voice settings and the input text

- Sends a POST request to OpenAI's TTS endpoint

- Receives MP3 audio data and saves it locally

- Loads the MP3 as an AudioClip and plays it back on the NPC

📦 Sample payload:

```json
{
  "model": "tts-1",
  "voice": "coral",
  "input": "Of course! The apples are over here.",
  "response_format": "mp3"
}
```

🔊 Key method:

```csharp
public void Speak(string text)
{
    StartCoroutine(SendTTSRequest(text));
}
```

📤 Output:

- Plays audio via audioSource.Play()
- Optionally triggers a text bubble update in parallel

🔗 Dependencies:

- UnityWebRequest

- AudioSource

- File I/O (WAV or MP3 saving/loading)

🧩 Module Integration Diagram (optional for visuals):

User (Voice or Button)

⬇

MicRecorder.cs → Whisper API

⬇

DialogueManager.cs

⬇

OpenAIChat.cs → ChatGPT

⬇

DialogueManager.cs

⬇

OpenAITTS.cs → TTS API

⬇

AudioSource.Play()

File Handling

Audio captured as AudioClip → converted to .wav (16-bit PCM, 44.1kHz mono)

WAV encoded using custom class (WavUtility or similar)

TTS audio received as MP3 → saved to StreamingAssets or Application.persistentDataPath
API Request Formats

6.1 Whisper

🧠 Speech-to-Text (STT) System — Whisper API Integration
Module: MicRecorder.cs

Overview

The Whisper integration enables the user to speak naturally into a microphone. Their voice is recorded as a WAV file and sent to OpenAI's /v1/audio/transcriptions endpoint. The resulting transcription is automatically passed to the dialogue system, allowing real-time, voice-driven interaction with the NPC.

Microphone Recording — Capturing User Speech

The user initiates recording by calling StartRecording(), which uses Unity's Microphone API:

```
public void StartRecording()
{
    recordedClip = Microphone.Start(null, false, 5, 44100);
    Debug.Log("🎙️ Started recording...");
}
```

Records from the default microphone

Captures 5 seconds at 44.1 kHz, mono

The recorded clip is stored in a local AudioClip reference

Recording is stopped manually by calling:

```
public void StopRecordingAndSend()
{
    Microphone.End(null);
    // ...
}
```

WAV File Encoding

Once recording stops, the AudioClip is serialized to a WAV file. This is done using a helper method SaveWav:

```
void SaveWav(string filePath, AudioClip clip)
{
    var samples = new float[clip.samples];
    clip.GetData(samples, 0);

    byte[] wav = WavUtility.FromAudioClip(clip, samples,
clip.channels, clip.frequency);
    File.WriteAllBytes(filePath, wav);
    Debug.Log("✅ WAV saved: " + filePath);
}
```

Extracts raw float samples from the clip

Converts the audio to WAV format

Saves the file at: Application.persistentDataPath/recorded.wav

Note: This WAV file is required by the Whisper API.

Whisper API Request — Sending Audio for Transcription

The coroutine SendToWhisper handles the POST request to OpenAI:

```
private IEnumerator SendToWhisper(string path)
{
    byte[] audioData = File.ReadAllBytes(path);

    WWWForm form = new WWWForm();
    form.AddBinaryData("file", audioData, "recording.wav",
"audio/wav");
    form.AddField("model", "whisper-1");

    UnityWebRequest www = UnityWebRequest.Post("https://api.openai.com/v1/audio/transcriptions", form);
    www.SetRequestHeader("Authorization", "Bearer " + apiKey);

    yield return www.SendWebRequest();
}
```

Request details:

Endpoint: /v1/audio/transcriptions

Content-Type: multipart/form-data

Fields:

file (WAV)

model = whisper-1

Authorization: Bearer {apiKey}

The response is a JSON object that includes the transcribed text.

Response Handling and Callback

After the response is received:

```
string json = www.downloadHandler.text;
WhisperResponse parsed = JsonUtility.FromJson<WhisperResponse>(json);
```

Then the parsed text is forwarded into the dialogue system:

```
if (!string.IsNullOrEmpty(parsed.text))
{
    dialogueManager.SendUserSpeech(parsed.text); // Launches the AI response
}
```

This triggers the same process as a button-activated prompt.

Output Example

Voice input: “Where can I find the apples?”
↓
Whisper returns: "Where can I find the apples?"
↓
DialogueManager sends this to ChatGPT → NPC replies via TTS

Class Structure

Public Methods:

StartRecording()

StopRecordingAndSend()

Private Coroutine:

SendToWhisper(string path)

Nested Class:

WhisperResponse for JSON parsing

Dependencies:

AudioClip, Microphone API, UnityWebRequest

WavUtility (custom or third-party utility for WAV export)

🛡️ Security Note

The API key is stored in a serialized string field

The WAV file is stored locally but deleted/overwritten after each use (or should be)

Recording requires mic permission on Android/Quest 2

6.2 TTS

🗣️ Text-to-Speech (TTS) System — Technical Description
Module: OpenAITTS.cs

Overview

The TTS system is responsible for transforming the NPC's response (generated by ChatGPT) into spoken audio using OpenAI's /v1/audio/speech endpoint. The resulting speech is played back through an AudioSource component in Unity, making the interaction feel more lifelike and accessible.

Entry Point: Calling the Speak() Method

The public method Speak(string text) is used to trigger voice synthesis. It accepts a string (typically a ChatGPT reply) and starts an asynchronous coroutine to handle the TTS process.

Example:

```
public void Speak(string text)
{
    StartCoroutine(SendTTSRequest(text));
}
```

Composing the TTS Request (JSON Body)

Within the coroutine SendTTSRequest, the text is formatted as a JSON payload. The payload includes the following fields:

model: "tts-1"

input: the raw text to convert

voice: one of OpenAI’s supported voices (e.g. "coral")

instructions: optional styling (e.g., tone or mood)

response_format: "mp3"

Snippet:

```
string jsonBody = $@"
{{
    ""model"": ""tts-1"",
    ""input"": ""{inputText}"",
    ""voice"": ""coral"",
    ""instructions"": ""Speak in a cheerful and positive tone."",
    ""response_format"": ""mp3""
}}";
```

This JSON string is then encoded as a byte array:

```
byte[] bodyRaw = Encoding.UTF8.GetBytes(jsonBody);
```

Sending the HTTP Request

A UnityWebRequest is created to POST the payload to OpenAI’s TTS endpoint:

```
string url = "https://api.openai.com/v1/audio/speech";

UnityWebRequest request = new UnityWebRequest(url, "POST");
request.uploadHandler = new UploadHandlerRaw(bodyRaw);
request.downloadHandler = new DownloadHandlerBuffer();
request.SetRequestHeader("Authorization", $"Bearer {apiKey}");
request.SetRequestHeader("Content-Type", "application/json");
```

🛡️ CertificateHandler is overridden in development to bypass SSL issues:

```
request.certificateHandler = new BypassCertificate();
```

Handling the Response

Once the server responds, the received MP3 binary is written to disk:

```
byte[] audioData = request.downloadHandler.data;
string path = Path.Combine(Application.persistentDataPath,
"speech.mp3");
File.WriteAllBytes(path, audioData);
```

Playing the Synthesized Audio

The saved MP3 is then loaded into Unity using UnityWebRequestMultimedia:

```
UnityWebRequest wwwAudio =
UnityWebRequestMultimedia.GetAudioClip("file://" + path,
AudioType.MPEG);
```

If successful, the AudioClip is assigned to the NPC's AudioSource and played:

```
AudioClip clip = DownloadHandlerAudioClip.GetContent(wwwAudio);
audioSource.clip = clip;
audioSource.Play();
```

Error Logging and Debugging

If the API call fails, error codes and response text are logged:

```
if (request.result != UnityWebRequest.Result.Success)
{
    Debug.LogError($"TTS Request Error: {request.responseCode} -
{request.error}");
    Debug.LogError("Server response: " +
request.downloadHandler.text);
}
```

Security Considerations

The API key is injected via a serialized field and passed in the Authorization header:

```
request.SetRequestHeader("Authorization", $"Bearer {apiKey}");
```

A local daily rate limiter can be added to restrict overuse (see RateLimiter.cs in other parts of the system). The certificate bypass should be disabled in production builds.

Security

A local rate-limiting mechanism enforces a daily cap on API queries to prevent abuse of the shared key.

A feature for interact to the NPC can use the microphone to record their own voice and send it to the Whisper API for transcription, allowing for personalized interactions. The user has also use button prompts to interact with the NPC, which sends predefined text to the ChatGPT API for generating responses.
The data isn't stored permanently, and the application does not collect any personal information from users.

For this project, the key is hidden for local development and testing purposes. In production, it should be securely stored and accessed via environment variables or secure vaults. The file EnvLoader.cs is designed to load the API key from a secure location, such as a local file or environment variable, to avoid hardcoding sensitive information in the source code. the file with the key is on .env file, which is not included in the repository to prevent accidental exposure with the .gitignore file.

Extensibility

Dialogue context can be expanded with memory (future)

Additional NPCs can reuse DialogueManager with different prompts

UI and scene templates can be swapped for other environments (school, home, etc.)

Localization framework can be added for multilingual use