Hochschule Aalen

Technische Dokumentation

Gestaltung von Chatbot-Interfaces und Avataren

Projektteam

Name	Matrikel-Nr.	E-Mail
Jan Herbst	82784	82784@studmail.htw-aalen.de
Gabriel Roth	82798	82798@studmail.htw-aalen.de
Florian Merlau	81775	81775@studmail.htw-aalen.de

Aalen, den 13. Januar 2026

Inhaltsverzeichnis

1 Kurzbeschreibung

Ziel dieses Projekts ist die Entwicklung einer webbasierten Anwendung zur interaktiven Kommunikation mit einem KI-gestützten virtuellen Agenten. Die Anwendung kombiniert eine Chatoberfläche mit Audio- und Videoausgabe um eine möglichst natürliche Mensch–Computer-Interaktion zu ermöglichen.

Der Benutzer kann über eine Texteingabe mit einem KI-Chatbot kommunizieren. Die eingegebenen Nachrichten werden an ein lokal betriebenes Sprachmodell über die Ollama-API gesendet. Die generierten Antworten werden sowohl textuell dargestellt als auch für eine Sprachsynthese aufbereitet und an eine Unity-Anwendung übertragen.

Zusätzlich wird die audiovisuelle Darstellung des virtuellen Charakters über WebRTC realisiert. Die Benutzeroberfläche ist übersichtlich gestaltet und erlaubt unter anderem die Auswahl verschiedener Charaktere sowie eine flexible Anpassung des Layouts.

2 Architektur

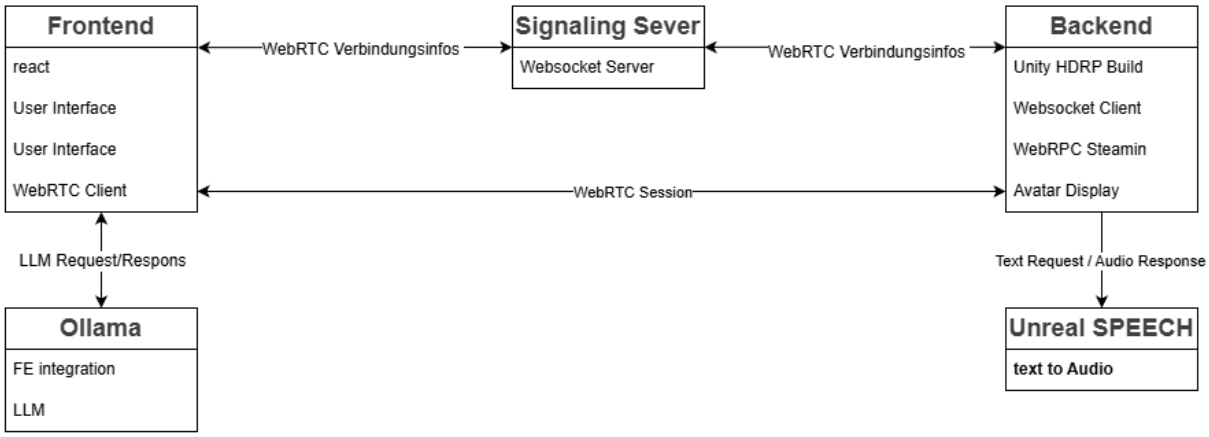

Abbildung 1: Architektur Diagramm

In diesem Abschnitt wird die Systemarchitektur detailliert dargestellt. Hier wiedie Interaktion beschrieben zwischen den verschiedenen Komponenten (Frontend, Backend, Datenbank und KI-Modelle).

3 Frontend

3.1 Beschreibung der Benutzeroberfläche

Die Anwendung ist als zweigeteilte Weboberfläche aufgebaut und folgt dem in der schriftlichen Ausarbeitung festgelegten Chat Interface-Konzept. Ziel ist eine intuitive Interaktion zwischen Benutzer und einem KI-gestützten virtuellen Argenten mit Text-, Audio- und Videoausgabe.

3.1.1 Gesamtlayout

Die Benutzeroberfläche besteht aus zwei Hauptbereichen:

- Linker Bereich: Chat- und Texteingabe
- Rechter Bereich: Avatar Video-Ausgabe

Beide Bereiche sind als React Card Komponenten umgesetzt.

3.1.2 Linker Bereich: Chat- und Texteingabe

Der linke Bereich ist die zentrale Kommunikationsschnittstelle.

Chatverlauf

- Chat der Konversation zwischen Benutzer und KI
- Automatisches Scrollen bei neuen Nachrichten
- Typing Indicator um die laufende KI-Antwort zu visualisieren

Textinput

- Eingabefeld am unteren Rand des Chatbereichs
- Send Button

Modell-Statusanzeige

- Falls das benötigte KI-Modell nicht lokal vorhanden ist erfolgt ein automatischer Download

3.1.3 Rechter Bereich: Avatar Video-Ausgabe

Der rechte Bereich dient der visuellen Darstellung des virtuellen Argenten und dessen Konfiguration.

Videoanzeige

- Anzeige eines Live-Video-Streams über ein <video>-Element
- Das Video wird über WebRTC bereitgestellt
- Responsive Darstellung mit object-fit: cover

Audiofreigabe

- Über den “Allow Audio” Button kann das Audio des virtuellen Agenten aktiviert werden
- Gängige Sicherheitsmechnismen des Browsers haben das Audio standarmäßig deaktiviert daher ist der Button notwendig

Charakterauswahl

- Dropdown-Menü zur Auswahl verschiedener Charaktere

- Auswahl wird per WebSocket an das Backend übermittelt
- Charakter 1 und 2: cartoonhafte Charaktere
- Charakter 3: realistischer Charakter
- Charakter 4: vollständige Verkörperung des realistischen Charakters

Größenanpassung

- Der rechte Bereich ist horizontal skalierbar
- Drag-Handle am linken Rand des rechten Bereichs
- Minimale und maximale Breite sind begrenzt um das vorgegebene Inerface Layout ein zu halten

3.2 Benutzerführung

1. Der Benutzer öffnet die Anwendung.
2. Das System überprüft automatisch, ob das benötigte KI-Modell vorhanden ist.
3. Falls erforderlich wird der Download des KI-Modells gestartet.
4. Der Benutzer gibt eine Nachricht in das Eingabefeld ein.
5. Die Nachricht wird:
    - im Chat angezeigt
    - an das KI-Backend (Ollama) gesendet
6. Die KI-Antwort wird:
    - im Chat angezeigt
    - per WebSocket an die Unity-Anwendung übertragen

3.3 Verwendete Web-Technologien

3.3.1 Frontend

- React Komponentbasierte Architektur
- Material UI (MUI) für UI-Komponenten wie bspw. Cards, Buttons, TextFields, Selects und weiteren Layout Elementen
- CSS für die Design OPtimierung

3.3.2 Kommunikation

WebSockets

- Echtzeitkommunikation mit Signaling Server und Unity Client
- Übertragung von Text

WebRTC

- Audio- und Video-Streaming

REST-API

- Kommunikation mit der lokalen Ollama REST-API

3.3.3 KI-Chatbot Integration

Ollama API

- Nutzung des lokal laufenden Sprachmodells Ollama3.1
- Chat-basierte Interaktion mit Kontextverlauf
- System-Prompt zur Definition des KI-Verhaltens

4 Chatbot

Die Implementierung des Chatbots basiert auf einer hybriden Architektur, die Next.js API-Routes als zentralen Vermittler (Proxy) nutzt. Dies ermöglicht eine strikte Trennung zwischen dem Frontend (Client) und den verschiedenen Backend-Diensten (Ollama: 11434, Python-Backend: 8000).

4.1 Architekturübersicht

Die Integration wurde in zwei logische Pfade unterteilt, um unterschiedlichen Anforderungen an Latenz und Funktionalität gerecht zu werden:

1. **Pfad A: Modell-Verwaltung (Direct Proxy Pattern)**
   Für administrative Aufgaben wie das Auflisten oder Herunterladen von Modellen kommuniziert der Next.js Server direkt mit der Ollama-API. Hierbei wird kein zwischengeschaltetes Backend benötigt, da keine komplexe Geschäftslogik (wie TTS) erforderlich ist.
2. **Pfad B: Chat-Interaktion (Orchestration Pattern)**
   Für die eigentliche Konversation fungiert die Next.js API als Orchestrator. Sie ruft sequenziell das Python-Backend auf, um erst die Text-Antwort (LLM Inferenz) und anschließend die Audio-Synthese (TTS) zu generieren, bevor ein gebündeltes Antwort-Paket an den Client zurückgeht.

4.2 Technische Detail-Implementierung

4.2.1 1. Modell-Synchronisation mit Streaming (Pfad A)

Die Route `api/models/route.ts` implementiert einen transparenten Proxy zu Ollama. Eine besondere technische Herausforderung war hierbei das *Pulling* (Herunterladen) neuer Modelle, da dieser Vorgang mehrere Minuten dauern kann. Hierfür wurde ein **ReadableStream** implementiert, der die `ndjson` (Newline Delimited JSON) Status-Updates von Ollama direkt an den Client durchreicht. Dies ermöglicht eine Echtzeit-Prozentanzeige im Frontend.

Code-Auszug der Streaming-Implementierung (`models/route.ts`):

```
// Weiterleitung des Ollama-Streams an den Client
const stream = new ReadableStream({
    async start(controller) {
        const reader = response.body.getReader();
        while (true) {
            const { done, value } = await reader.read();
            if (done) break;
            // Chunks unverändert durchreichen
            controller.enqueue(value);
        }
        controller.close();
    }
});
return new Response(stream, {
    headers: { 'Content-Type': 'application/x-ndjson' }
});
```

4.2.2 2. Chat-Orchestrierung (Pfad B)

Die Route `api/chat/route.ts` übernimmt die Steuerung des Gesprächsflusses. Anstatt dass der Client mehrere Anfragen stellen muss, bündelt der Server diese Logik (Backend-for-Frontend Pattern).

Ablauf eines Requests:

- **Schritt 1: Inferenz.** Der Server sendet den User-Prompt und die Historie an das Python-Backend (`POST :8000/chat`). Dort wird Ollama bzw. das LLM angefragt.
- **Schritt 2: Fallback.** Sollte das Backend nicht erreichbar sein, fängt der Next.js Server den Fehler ab und generiert eine statische Fehlerantwort ("Brain server unavailable"), um Abstürze im Frontend zu verhindern.
- **Schritt 3: Synthese (TTS).** Liegt eine gültige Textantwort vor, sendet der Next.js Server diesen Text sofort an den TTS-Endpunkt (`POST :8000/tts`).
- **Schritt 4: Aggregation.** Die Audio-Daten (MP3 Base64) und der Text werden in einem einzigen JSON-Objekt zusammengeführt und final an den Client gesendet.

Dieser Ansatz reduziert die Anzahl der Round-Trips für den Client drastisch und synchronisiert die Verfügbarkeit von Text und Audio.

Implementierung der Orchestrierung (`chat/route.ts`):

```
// 1. Inferenz anfragen
const chatResponse = await fetch(`${API_BASE_URL}/chat`, { ... });
const chatData = await chatResponse.json();

// 2. Audio generieren (falls Text vorhanden)
if (chatData.text) {
    const ttsResponse = await fetch(`${API_BASE_URL}/tts`, {
        body: JSON.stringify({ text: chatData.text, ... })
    });
    // ... Buffer zu Base64 konvertieren
}
```

4.2.3 3. Prompt Engineering & Ollama Schnittstelle

Die Kommunikation mit Ollama erfolgt über standardisierte HTTP-POST-Requests. Da Ollama eine OpenAI-kompatible API anbietet, ist der Wechsel auf andere Modelle oder Provider theoretisch nahtlos möglich.

Verwendetes Modell Standardmäßig wird das Modell `llama3.1:latest` verwendet. Dieses wurde gewählt, da es eine gute Balance zwischen Latenz und Antwortqualität für eine Echtzeit-Konversation bietet.

Prompt-Struktur (JSON Payload) Der Prompt wird nicht als reiner Text, sondern als strukturiertes JSON-Objekt an die API gesendet. Das Backend konstruiert hierfür einen *System-Prompt*, um die Persona des Avatars zu definieren, und fügt die Chat-Historie als Kontext an.

Struktur des Requests (an `/api/generate` oder `/api/chat`):

```
{
  "model": "llama3.1:latest",
  "messages": [
    {
      "role": "system",
      "content": "Du bist ein hilfsbereiter KI-Assistent in einem interaktiven Setup..."
    },
    {
      "role": "user",
      "content": "Hallo, wie geht es dir?"
    }
  ],
  "stream": false,
  "options": {
    "temperature": 0.7
  }
}
```

Datenfluss-Diagramm Die nachfolgende Grafik illustriert den Weg einer Benutzeranfrage durch die Systemarchitektur – von der Eingabe im Frontend bis zur audiovisuellen Ausgabe.

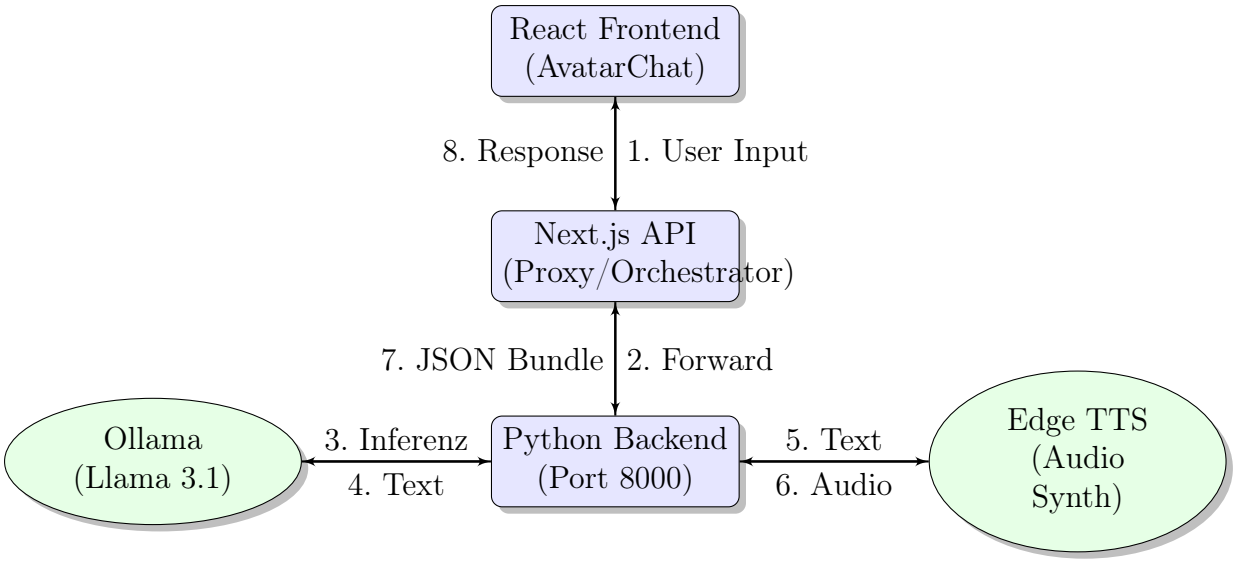

Abbildung 2: Datenfluss der Chatbot-Anfrage (Orchestration Pattern)

4.2.4 4. Interne Kommunikation (Python Backend)

Die Kommunikation zwischen dem Next.js Proxy und dem Python-Backend (auf Port 8000) erfolgt über spezialisierte Endpunkte. Hier werden die Aufgaben in atomare Schritte zerlegt.

Endpunkt 1: `POST /chat` Dieser Endpunkt nimmt den bereinigten Text entgegen und interagiert mit dem LLM.

```
// Request (Next.js -> Python)
{
  "message": "Erzähl mir einen Witz.",
  "history": [
    {"role": "user", "text": "Hi"},
    {"role": "bot", "text": "Hallo!"}
  ],
  "model": "llama3.1:latest"
}

// Response (Python -> Next.js)
{
  "text": "Warum können Seeräuber...",
  "emotion": "Happy",
  "usage": { "tokens": 45 }
}
```

Endpunkt 2: `POST /tts` Falls die Antwort Text enthält, wird dieser sofort an den TTS-Dienst gesendet.

```
// Request (Next.js -> Python)
{
```

```
  "text": "Warum können Seeräuber den Kreisumfang nicht berechnen?",
  "language": "de",
  "emotion": "happy"
}

// Response (Python -> Next.js)
{
  "audio_base64": "SUQzBAAAAAAAI1RTU0U...", // MP3 Daten
  "duration": 3.5
}
```

4.2.5 5. Frontend-Integration (Chat-Interface)

Während das allgemeine Frontend-Design in Abschnitt 3 behandelt wird, liegt die Logik der Chat-Interaktion (File: `AvatarChat.tsx`).

Komponenten-Logik Die `AvatarChat`-Komponente fungiert als zentraler Controller für die Interaktion. Sie verwaltet den Zustand der Konversation und synchronisiert sich mit den APIs. Wichtige Hooks und States:

- `useState<Message[]>`: Speichert den lokalen Chat-Verlauf.
- `useEffect`: Lädt beim Start die Historie aus dem `localStorage` und prüft die Verfügbarkeit der Modelle.
- `useRef`: Dient dem automatischen Scrollen zum neuesten Nachrichteneintrag (Äuto-Scroll").

State-Management Flow Das nachfolgende Diagramm zeigt den internen Ablauf innerhalb der React-Komponente beim Absenden einer Nachricht.

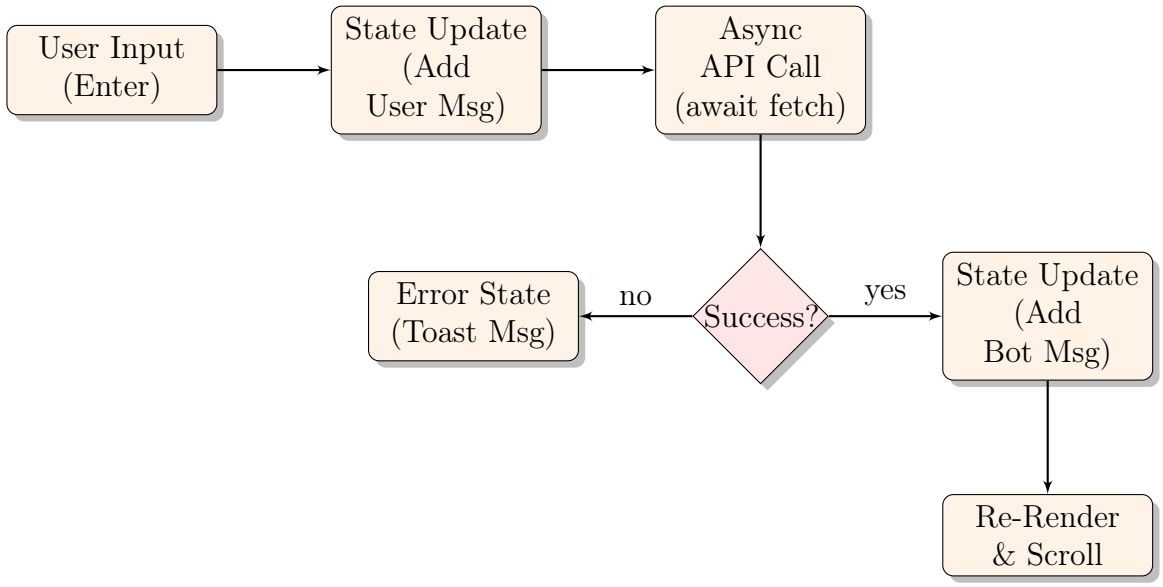

Abbildung 3: React State-Flow (Senden einer Nachricht)

5 Backend

Im Backend haben wir ein Unity HDRP-Projekt, in dem der Avatar dargestellt wird. Um den Avatar im Frontend anzuzeigen, nutzen wir WebRTC, um das Unity-Projekt an das Frontend zu streamen. Damit eine WebRTC-Session aufgebaut werden kann, wird ein Websocket-Signaling-Server verwendet. Das Backend verbindet sich mit diesem Server und erhält alle WebRTC-Anfragen vom Frontend. Diese Anfragen enthalten die Verbindungsinformationen. Anschließend wird eine Verbindung aufgebaut und eine WebRTC-Session gestartet.

Der zweite Teil betrifft die Sprachsynthese des Avatars: Sobald das Backend über die Websocket-Verbindung die Antwort des LLM erhält, wird der Text an Unreal Speech weitergeleitet. Dort wird aus dem Text eine Audiodatei erzeugt und an das Backend zurückgesendet. Danach wird das Audio abgespielt, sodass man hören kann, was der Avatar sagt.

Für die Lippen-Synchronisation wird Oculus LipSync Verwendet. Dabei wird das aktuellen Audio-Signal auf 14 Gesichtsausdrücke gemappt. Jeder Charakter muss zuvor so eingerichtet sein, dass die entsprechenden Blendshape-IDs den jeweiligen Gesichtsausdrücken zugeordnet sind.

Zusätzlich verfügt das Backend über ein Script, das einen Charakterwechsel ermöglicht, sodass immer der aktuell ausgewählte Avatar angezeigt wird.