Multilinguales Text-to-Speech System mit LLM Integration

Lokale, sichere Sprachsynthese und KI-Dialoge
für natürliche Interaktionen

Amine Lahbib

Iheb Khamelia

Institution: Hochschule Kaiserslautern
Department of Applied Computer Science
Studienprojekt Bericht: TTS Project mit Rust
Betreuer: Prof. Norbert Rösch
Dozent: Prof.Thomas Allweyer

Kaiserslautern, November 2025

Zusammenfassung

Das TTS-Projekt ist eine leistungsstarke, datenschutzfreundliche und vollständig lokal ausgeführte Plattform für Text-to-Speech (TTS), KI-basierte Chats und sprachgestützte Interaktionen. Durch die Integration von Piper TTS für die Sprachsynthese und Ollama für die KI-Konversation bietet das System eine realistische Sprachwiedergabe, natürliche Dialogführung und sichere, lokal verarbeitete Interaktionen.

Inhaltsverzeichnis

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Executive Summary

Das TTS-Projekt ist eine leistungsstarke, mehrsprachige Text-to-Speech- und KI-Chat-Plattform, die in Rust implementiert ist. Es kombiniert Piper TTS für die Sprachsynthese mit Ollama für konversationelle KI und stellt damit eine vollständige, sprachgesteuerte Assistenzlösung bereit.

Was das System leistet:

- Konvertiert Text in acht Sprachen in natürlich klingende Sprache
- Bietet KI-gestützte Chat-Konversationen
- Ermöglicht sprachbasierte Interaktionen mit Echtzeit-Streaming
- Stellt eine übersichtliche Weboberfläche für alle Funktionen bereit

Wichtigste Ergebnisse:

- TTS-Latenz im Sub-Sekunden-Bereich durch intelligentes Caching
- Echtzeit-Streaming von Antworten über WebSocket
- Produktionsreife Architektur mit umfassender Fehlerbehandlung
- Docker-basierte Bereitstellung für einfache Inbetriebnahme

Warum das relevant ist: Das System richtet sich an Entwicklerinnen und Entwickler sowie Endnutzer, die eine schnelle, lokale und datenschutzfreundliche Sprach-KI benötigen, ohne auf Cloud-Dienste angewiesen zu sein. Alle Komponenten laufen vollständig auf dem eigenen System.

2 Project Overview

2.1 Purpose

Dieses Projekt deckt drei zentrale Anwendungsfälle ab:

1. **Text-to-Speech**: Beliebigen Text in mehreren Sprachen und mit verschiedenen Stimmen in Sprache umwandeln
2. **AI Chat**: Gespräche mit einem lokalen LLM führen (ohne Cloud-Anbindung)
3. **Voice Chat**: Beide Ansätze kombinieren – per Chat mit der KI interagieren und gesprochene Antworten erhalten

2.2 Architecture at a Glance

Das System folgt einer einfachen Client-Server-Architektur:

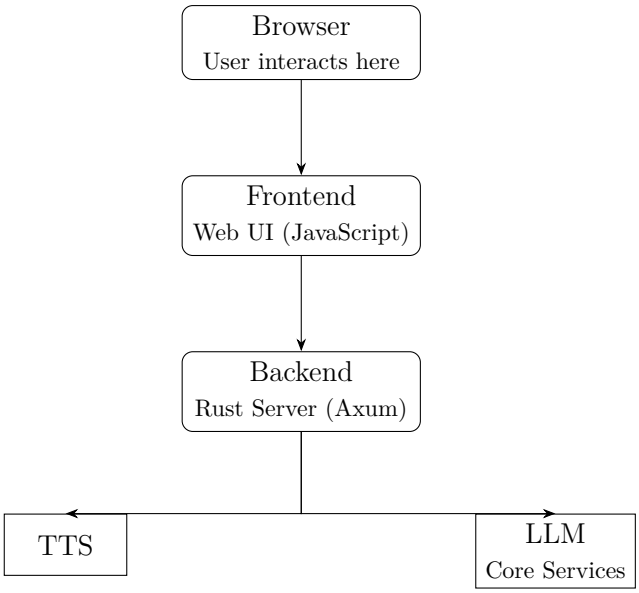

Abbildung 1: High-level system architecture

2.3 Supported Languages

- Englisch (US) – 6 Stimmen
- Deutsch – 3 Stimmen
- Französisch – 2 Stimmen
- Spanisch – 2 Stimmen
- Italienisch – 2 Stimmen
- Niederländisch – 3 Stimmen
- Ukrainisch – 1 Stimme

3 Tech Stack

3.1 Backend

Komponente	**Technologie**	**Begründung**
Sprache	Rust	Schnell, sicher, ideal für Hochleistungs-Services
Web-Framework	Axum 0.8	Modernes asynchrones HTTP-/WebSocket-Framework
Async Runtime	Tokio	Multithreaded asynchrone Ausführung
TTS-Engine	Piper-rs	ONNX-basierte neuronale TTS
LLM-Client	reqwest	HTTP-Client für die Kommunikation mit Ollama
Vektor-Datenbank	Qdrant	Optionale persistente Ablage von Konversationen

3.2 Frontend

Komponente	Technologie	Begründung
Sprache	JavaScript (ES6+)	Moderne Browserunterstützung
Framework	Vanilla JS	Kein Build-Schritt, schnelle Ladezeiten
Styling	CSS3	Schlankes, responsives Design
Audio	Web Audio API	Native Audio-Wiedergabe im Browser

3.3 Infrastructure

- **Docker** + **Docker Compose** – Einfache Bereitstellung
- **Ollama** – Lokaler LLM-Server
- **Qdrant** – Optionale Vektor-Datenbank für Konversationen

4 Component Breakdown

4.1 Backend

Das Backend ist in drei zentrale Crates (Rust-Pakete) aufgeteilt:

4.1.1 Server Module (`server/`)

Aufgabe: Verarbeitung aller HTTP-/WebSocket-Anfragen und Weiterleitung an die entsprechenden Dienste.

Wesentliche Verantwortlichkeiten:

- Routing und Validierung von Anfragen
- Rate Limiting (60 Anfragen pro Minute)
- CORS-Behandlung
- Metrik-Erfassung
- Fehlerbehandlung

Zentrale Bestandteile:

```
server/
+-- main.rs          # Entry point, route definitions
+-- config.rs        # Configuration management
+-- metrics.rs       # Performance tracking
+-- validation.rs    # Input validation
\-- error.rs         # Error types
```

Architektur:

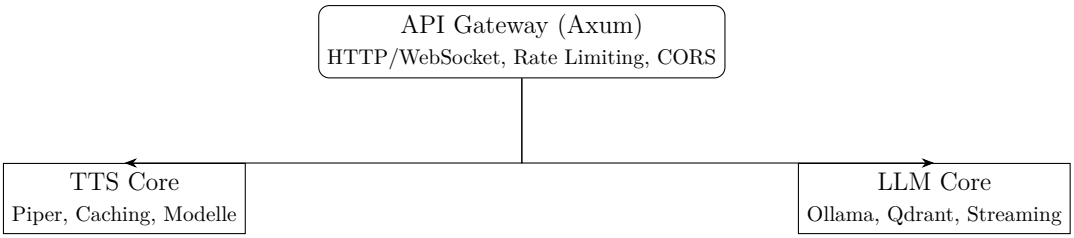

Abbildung 2: Serverarchitektur: API-Gateway und Cores

4.1.2 TTS Core Module (`tts_core/`)

Aufgabe: Verwaltung der Text-to-Speech-Synthese mit Piper-TTS-Modellen.
Wichtige Funktionen:

- Modell-Caching (bis zu 15 Modelle im Speicher)
- Antwort-Caching (500 Einträge, Lebensdauer 1 Stunde)
- Einfügen natürlicher Pausen
- Mehrere Stimmen pro Sprache

Ablauf:

1. Empfang von Text plus Sprach-/Stimmenauswahl
2. Prüfung des Caches (schneller Pfad)
3. Falls kein Treffer: Modell laden und Audio synthetisieren
4. Einfügen von Pausen an Satzzeichen
5. Kodierung nach WAV und Rückgabe als Base64

Performance:

- Cache-Treffer: < 10 ms
- Cache-Fehlgriff: 100–500 ms (abhängig von der Textlänge)

4.1.3 LLM Core Module (`llm_core/`)

Aufgabe: Verwaltung von Konversationen mit dem lokalen LLM (Ollama).
Wichtige Funktionen:

- Verwaltung des Gesprächsverlaufs
- Streaming von Antworten Token für Token
- Optionale Persistenz in Qdrant
- Antwort-Caching

Ablauf:

1. Empfang einer Nachricht inklusive Konversations-ID
2. Laden des Konversationsverlaufs (typisch die letzten vier Turns)
3. Senden der Anfrage an Ollama und Empfangen der Antwort
4. Streaming der Tokens zurück an den Client
5. Aktualisierung des Konversations-Caches

Performance:

- Erstes Token: < 100 ms (Streaming)
- Gesamte Antwort: 1–10 s (abhängig von der Komplexität)

4.2 Frontend

Das Frontend ist als Single-Page-Application mit modularer Architektur aufgebaut:

4.2.1 Application Structure

Haupteinstiegspunkt: `js/main.js`

Struktur:

```
frontend/
+-- index.html          # Main HTML structure
+-- js/
|   +-- main.js         # App initialization
|   +-- components/     # Reusable UI components
|   +-- services/       # API & WebSocket clients
|   +-- tabs/           # Tab-specific logic
|   \-- utils/          # Helper functions
+-- css/                # Stylesheets
\-- tabs/               # HTML templates
```

Architektur:

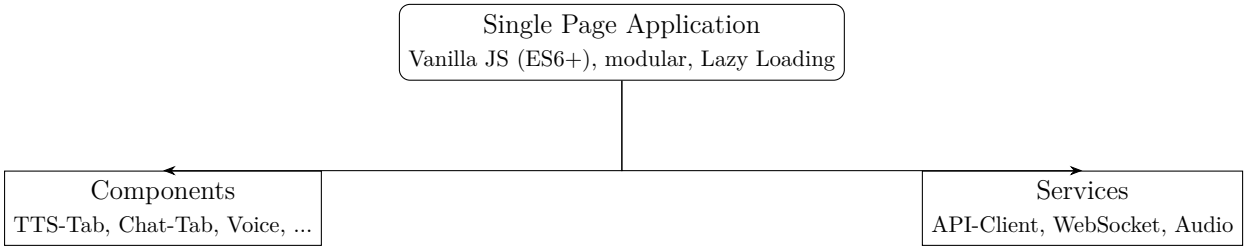

Abbildung 3: Frontend-Architektur: SPA, Komponenten und Services

4.2.2 Key Components

TTS-Tab (`tabs/tts.js`): Texteingabefeld, Auswahl von Sprache und Stimme, Audiowiedergabe-Steuerung, Spektrogramm-Visualisierung, Download-Funktion

Chat-Tab (`tabs/chat.js`): Ein- und Ausgabe von Nachrichten, Verlauf der Konversation, Unterstützung für Streaming (Tokens in Echtzeit), Markdown-Rendering

Voice-Chat-Tab (`tabs/voice-chat.js`): Kombination aus Chat und TTS, Audiowiedergabe der Antworten, Fortlaufender Gesprächskontext

Server-Tab (`tabs/server.js`): Anzeige von Echtzeit-Metriken, Systeminformationen, Performance-Diagramme

4.2.3 Services

API-Service (`services/api.js`):

- Verarbeitung von HTTP-Anfragen
- Antwort-Caching (5 Minuten TTL)
- Fehlerbehandlung und Wiederholungslogik
- Deduplizierung identischer Anfragen

WebSocket-Service (`services/websocket.js`):

- Verwaltung der WebSocket-Verbindung
- Verarbeitung von Streaming-Nachrichten
- Automatischer Wiederaufbau der Verbindung

Voice-Service (`services/voice.js`):

- Hilfsfunktionen für Audiowiedergabe
- Vorbereitung für zukünftige Audioaufnahmen

5 User Guide

5.1 TTS-Tab

Zweck: Text in jeder unterstützten Sprache in Sprache umwandeln.
Bedienung:

1. **Text eingeben** in das Texteingabefeld
2. **Sprache auswählen** im Dropdown (z. B. „en_US“ für Englisch)
3. **Stimme wählen** (optional – sonst wird die Standardstimme genutzt)
4. **Auf „Synthesize“ klicken**

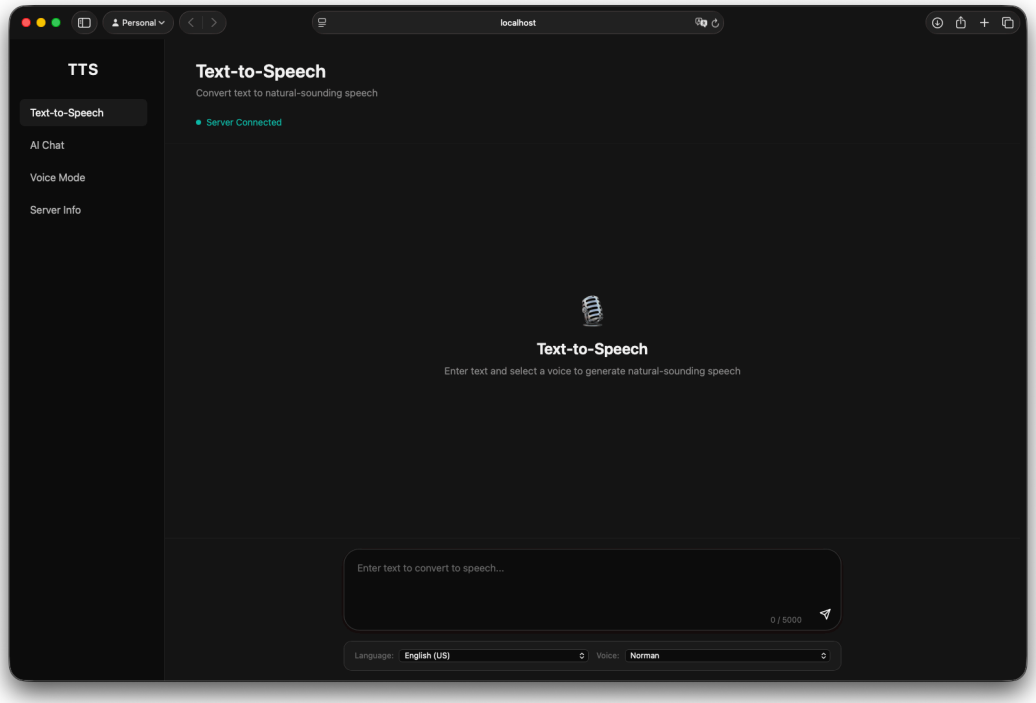

Abbildung 4: Benutzeroberfläche des TTS-Tabs

5. **Audio anhören** über die Player-Steuerung
6. **Audio-Datei herunterladen**, falls benötigt

Funktionen:

- Echtzeit-Spektrogramm
- Mehrere Stimmen pro Sprache
- Audioplayer (Abspielen, Pausieren, Download)
- Unterstützung langer Texte (internes Chunking)

Tipps:

- Satzzeichen sorgen für natürlichere Pausen
- Unterschiedliche Stimmen haben unterschiedliche Charakteristika (z. B. Geschlecht, Klangfarbe)
- Durch Caching werden wiederholte Anfragen nahezu sofort beantwortet

5.2 AI-Chat-Tab

Zweck: Konversation mit einem lokalen KI-Assistenten führen.
Bedienung:

1. **Nachricht eingeben** in das Eingabefeld

2. **„Send“ klicken** oder Enter drücken
3. **Antwort beobachten**, die in Echtzeit (Token-Streaming) erscheint
4. **Konversation fortsetzen**; der Kontext wird automatisch mitgeführt

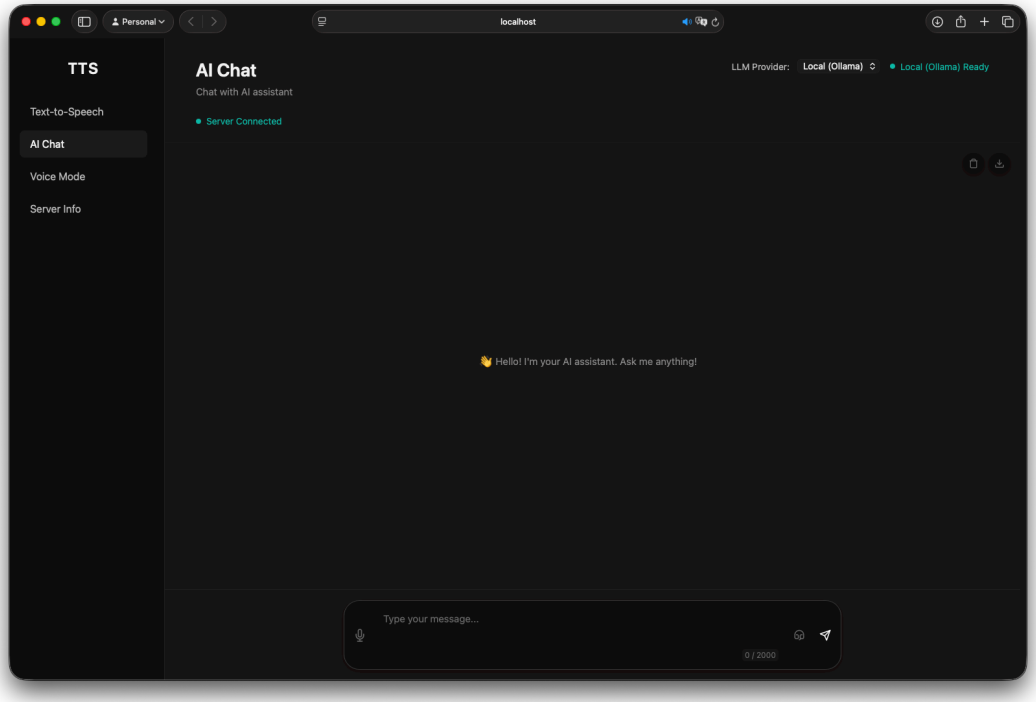

Abbildung 5: Benutzeroberfläche des AI-Chat-Tabs

Funktionen:

- Streaming-Antworten (Tokens erscheinen schrittweise)
- Konversationsverlauf (letzte Turns)
- Markdown-Unterstützung in Antworten
- Verwaltung von Konversations-IDs

Tipps:

- Die Konversation läuft weiter, bis explizit eine neue gestartet wird
- Streaming verbessert die wahrgenommene Reaktionszeit
- Rückfragen sind möglich, da der Kontext erhalten bleibt

5.3 Voice-Mode-Tab

Zweck: Per Chat mit der KI interagieren und gesprochene Antworten erhalten.

Bedienung:

1. **Sprache für die Sprachausgabe wählen**
2. **Stimme auswählen** (optional)
3. **Nachricht eingeben** und auf „Send“ klicken
4. **Gesprochene Antwort anhören**
5. **Textantwort parallel lesen**

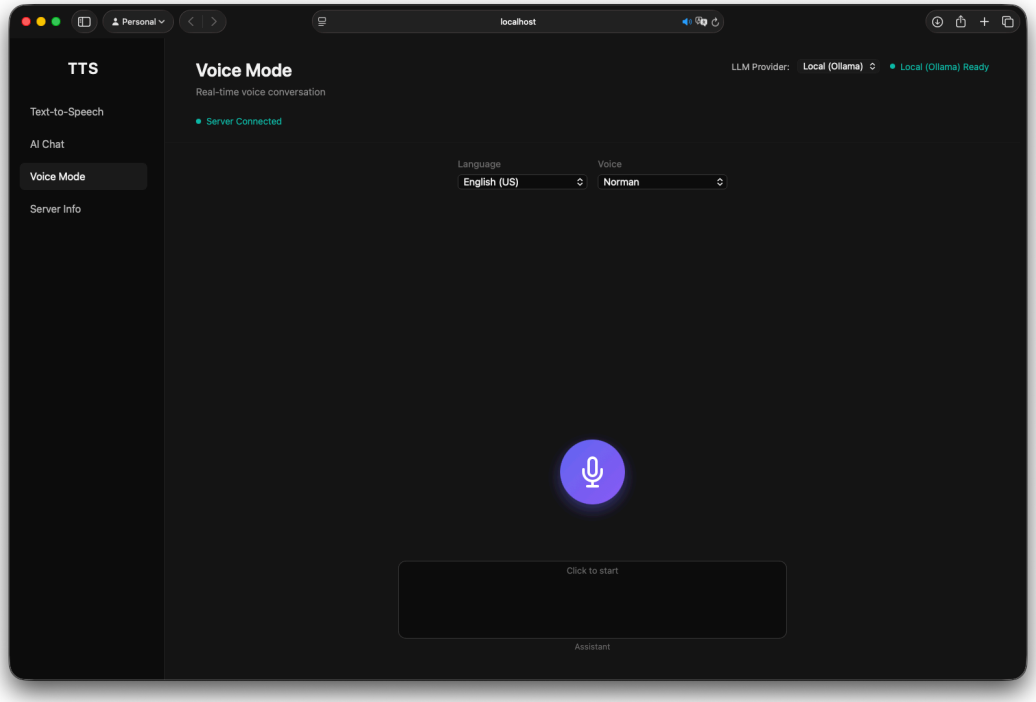

Abbildung 6: Benutzeroberfläche des Voice-Mode-Tabs

Funktionen:

- Kombinierte Text- und Audioantworten
- Fortlaufender Gesprächskontext
- Audiowiedergabe in Echtzeit nach Textgenerierung
- Anzeige von Originaltext und bereinigter Textvariante

Tipps:

- Besonders geeignet für freihändige Nutzung
- Audio wird nach dem Text generiert, daher leichte Zusatzlatenz
- Nützlich für Barrierefreiheit und Vorleseszenarien

5.4 Server-Info-Tab

Zweck: Überwachung der Serverleistung und des Systemzustands.

Angezeigte Inhalte:

- **Systemmetriken:**
  - CPU-Auslastung in Prozent
  - Speicherauslastung (belegt/gesamt)
  - Uptime
  - Systemlast
- **Anfrage-Statistiken:**
  - Gesamtanzahl der Anfragen
  - Anfragen pro Endpoint (TTS, Chat, Voice Chat)
  - Fehleranzahl
  - Durchschnittslatenz
- **TTS-Metriken:**
  - Anzahl der Synthesen
  - Cache-Hit-Rate
  - Durchschnittliche Synthesezeit
- **LLM-Metriken:**
  - Anzahl der Anfragen
  - Durchschnittliche Antwortzeit
  - Token-Statistiken
  - Anzahl der Timeouts

Bedienung:

1. **Server-Info-Tab öffnen**
2. **Metriken in Echtzeit verfolgen** (automatische Aktualisierung)
3. **Performance beobachten**, um Systemlast zu beurteilen
4. **Fehler prüfen**, falls etwas nicht funktioniert

Tipps:

- Hohe Cache-Hit-Rate deutet auf gute Performance hin
- Steigende Latenzen können auf Engpässe hinweisen
- Fehlerzähler helfen bei der Fehlersuche

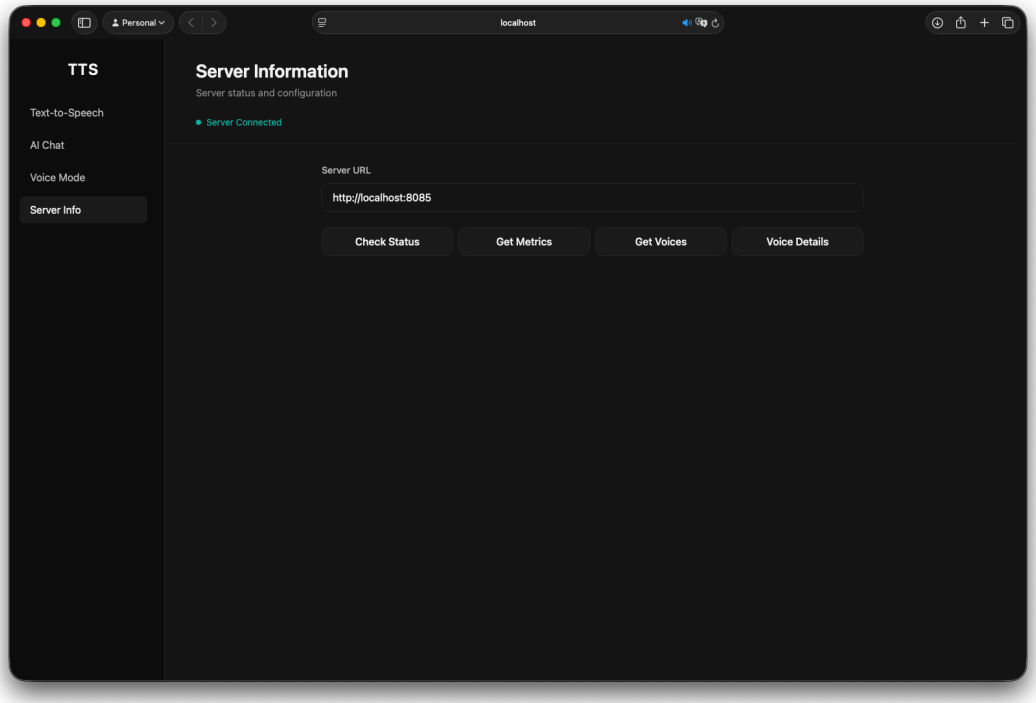

Abbildung 7: Benutzeroberfläche des Server-Info-Tabs

6 Appendices

6.1 Appendix A: API Endpoints

Health Check:

```
GET /health
-> Returns: "ok"
```

List Voices:

```
GET /voices
-> Returns: ["en_US", "de_DE", ...]
```

Text-to-Speech:

```
POST /tts
Body: {
  "text": "Hello",
  "language": "en_US",
  "voice": "norman"
}
-> Returns: {
  "audio_base64": "...",
  "duration_ms": 1234,
  "sample_rate": 22050
}
```

Chat:

```
POST /chat
Body: {
  "message": "Hello",
  "conversation_id": "uuid"
}
-> Returns: {
  "reply": "...",
  "conversation_id": "uuid"
}
```

Voice Chat:

```
POST /voice-chat
Body: {
  "message": "Hello",
  "language": "en_US",
  "voice": "norman"
}
-> Returns: {
  "reply": "...",
  "audio_base64": "...",
  "conversation_id": "uuid"
}
```

WebSocket Streaming:

```
GET /ws/chat/stream?message=Hello&language=en_US
-> WebSocket connection with streaming tokens
```

Metrics:

```
GET /metrics
-> Returns: system metrics (CPU, memory, requests)

GET /metrics/detailed
-> Returns: comprehensive metrics (per-endpoint, TTS, LLM)
```

6.2 Appendix B: Configuration

Environment Variables:

Variable	Default	Beschreibung
PORT	8085	Server-Port
LLM_MODEL	llama3	Name des Ollama-Modells
OLLAMA_BASE_URL	http://localhost:11434	Basis-URL des Ollama-Servers
RATE_LIMIT_PER_MINUTE	60	Anfrage-Limit pro Minute
RUST_LOG	info	Log-Level (debug, info, warn, error)

Docker Deployment:

```
docker compose up --build
```

Local Development:

```
cargo run --release -p server
```

6.3 Appendix C: Performance Benchmarks

TTS-Performance:

- Cache-Treffer: < 10 ms
- Cache-Fehlgriff: 100–500 ms
- Synthesegeschwindigkeit: ca. 10× Echtzeit

LLM-Performance:

- Erstes Token (Streaming): < 100 ms
- Durchschnittliche Antwort: 1–3 s
- Komplexe Anfragen: 5–10 s

System-Performance:

- Durchsatz: 60 Requests/Minute (Rate Limiting)
- Speicherbedarf: 500 MB bis 2 GB (abhängig von gecachten Modellen)
- CPU-Auslastung: 10–30 % (Normalbetrieb)

6.4 Appendix D: Troubleshooting

Problem: Modelle werden nicht geladen

- Prüfen, ob `models/map.json` vorhanden ist
- Sicherstellen, dass ONNX-Dateien im Verzeichnis `models/` liegen
- Datei- und Zugriffsrechte prüfen

Problem: Ollama-Verbindungsfehler

- Prüfen, ob Ollama läuft: `curl http://localhost:11434/api/tags`
- Wert der Umgebungsvariable `OLLAMA_BASE_URL` verifizieren
- Container-Status prüfen (falls Docker verwendet wird)

Problem: Hoher Speicherverbrauch

- Modellcache-Größe in `TtsManager` reduzieren
- Größe des Konversationscaches verringern
- Aggressiveres Entfernen selten genutzter Modelle aktivieren

Problem: Rate-Limit-Fehler

- `RATE_LIMIT_PER_MINUTE` bei Bedarf erhöhen
- Kurz warten und erneut versuchen
- Prüfen, ob mehrere Clients gleichzeitig Anfragen senden

7 FAZIT

7.1 Summary

Das TTS-Projekt stellt eine produktionsreife Plattform für mehrsprachige Text-to-Speech- und KI-Chat-Funktionen bereit. Mit Rust für die Performance und JavaScript für die Zugänglichkeit deckt es zentrale Anforderungen sprachbasierter Anwendungen ab.

7.2 Key Strengths

- **Performance**: Sub-Sekunden-TTS-Latenz durch intelligentes Caching
- **Datenschutz**: Vollständig lokale Ausführung ohne Cloud-Abhängigkeiten
- **Flexibilität**: Mehrere Sprachen, Stimmen und Nutzungsmodi
- **Entwicklerfreundlichkeit**: Klare Code-Struktur, gute Dokumentation, Docker-Support
- **Produktionsreife**: Umfassende Fehlerbehandlung, Metriken, Rate Limiting

7.3 Areas for Future Improvement

- **Authentifizierung**: Einführung von API-Keys oder JWT für produktive Umgebungen
- **Monitoring**: Erweiterte Metrik-Exports (z. B. Prometheus/Grafana)
- **Tests**: Umfassendere Testabdeckung im Frontend
- **Dokumentation**: Zusätzliche Beispiele und Schritt-für-Schritt-Anleitungen

7.4 Final Thoughts

Das Projekt demonstriert solide Software-Engineering-Praktiken mit Fokus auf Performance und Nutzererlebnis. Die Architektur ist klar strukturiert, der Code gut wartbar, und der Funktionsumfang deckt die wichtigsten Anwendungsfälle ab. Mit ergänzender Authentifizierung und erweitertem Monitoring ist die Lösung bereit für den produktiven Einsatz.

Die Kombination aus lokalem LLM (Ollama) und hochwertiger TTS (Piper) ergibt eine Lösung, die Privatsphäre respektiert und zugleich eine sehr gute Performance liefert.

Acknowledgments

Dieses Projekt verwendet Sprachmodelle aus dem **Hugging-Face**-Repository `rhasspy/piper-voices`. Diese Modelle werden unter der **MIT-Lizenz** bereitgestellt und müssen im Einklang mit den Lizenzbedingungen verwendet werden. Wir bedanken uns bei:

- **Hugging Face** und der **rhasspy/piper-voices**-Community für hochwertige mehrsprachige Sprachmodelle
- Den Teams hinter **Piper TTS** und **eSpeak-ng** für die exzellente TTS-Infrastruktur

- Dem **Ollama**-Team für zuverlässige lokale LLM-Tools
- Der **Rust**-Community für das starke Ökosystem an Crates
- Allen Open-Source-Beitragenden, die dieses Projekt ermöglicht haben

Lizenzkonformität: Alle in diesem Projekt verwendeten Sprachmodelle unterliegen der MIT-Lizenz und müssen entsprechend den Lizenzbedingungen eingesetzt werden. Nutzer sind dafür verantwortlich, die Einhaltung der Lizenz bei Verwendung oder Weiterverbreitung sicherzustellen.