FernUniversität in Hagen

– Fakultät für Mathematik und Informatik –

Lehrgebiet Datenbanken und Informationssysteme

Entwicklung eines KI-gestützten Spracherkennungssystems zur automatischen Transkription von Meetings

Fachpraktikum Softwareentwicklung mit Methoden der Künstlichen Intelligenz

vorgelegt von

Fabian Michael Scherer
Yolanda Hadiana Fiska
Pakize Gökkaya
Mike Wild

Betreuer : Dr.-Ing. Otmane Azeroual

INHALTSVERZEICHNIS

1 EINLEITUNG

Fabian Scherer

Die zunehmende Digitalisierung der Arbeits- und Kommunikationswelt führt zu einem stetig wachsenden Bedarf an effizienten Methoden zur Erfassung und Dokumentation von Informationen. Meetings und Konferenzen erzeugen eine große Menge an Gesprächsinhalten, deren manuelle Protokollierung jedoch zeitaufwendig, fehleranfällig und mitunter unvollständig ist. Moderne Verfahren der automatischen Spracherkennung (Automatic Speech Recognition, ASR) ermöglichen es, diesen Prozess zu automatisieren und somit die Nachverfolgbarkeit und Effizienz deutlich zu verbessern.

Durch den Einsatz künstlicher Intelligenz (KI) ist es inzwischen möglich, gesprochene Sprache in Echtzeit zuverlässig in Text zu überführen. Neuere Deep-Learning-Ansätze haben die Genauigkeit von Spracherkennungssystemen erheblich gesteigert und erlauben auch die Differenzierung verschiedener Sprecher sowie die Anpassung an Akzente oder Gesprächskontexte.

Fabian Scherer

Das vorliegende Projekt verfolgt das Ziel, ein KI-gestütztes Spracherkennungssystem zu entwickeln, das Meetings und Konferenzen automatisch transkribiert. Die gesprochene Sprache wird dabei in Echtzeit erkannt, in Text konvertiert und in einer Datenbank gespeichert. Die gespeicherten Protokolle sind durchsuchbar und können den Teilnehmern über einen QR-Code zur Verfügung gestellt werden. Zusätzlich ermöglicht die Integration einer lernfähigen Komponente, dass das System seine Erkennungsgenauigkeit im Laufe der Nutzung kontinuierlich verbessert.

Neben der Zeitersparnis bietet das System auch eine erhöhte Barrierefreiheit, da transkribierte Inhalte insbesondere für Personen mit Hörbeeinträchtigungen oder für nachträgliche Analysen zugänglich gemacht werden. Potenzielle Anwendungsfelder reichen von Unternehmen über den Bildungsbereich bis hin zum Journalismus, wo automatisierte Transkriptionen zur besseren Dokumentation und schnelleren Informationsverarbeitung beitragen können.

Damit leistet das Projekt einen Beitrag zur Weiterentwicklung moderner Kommunikationsunterstützungssysteme, indem es KI-basierte Spracherkennung nicht nur zur Transkription, sondern auch zur intelligenten Aufbereitung und Verfügbarkeit von Gesprächsinhalten nutzbar macht.

ANMERKUNG

Mike Wild

Der Quelltext der Anwendung ist unter dem folgenden Link zu finden: `https://github.com/Gareck88/SS2025FP_T2`

2 GRUNDLAGEN & STAND DER TECHNIK

Fabian Scherer

Die automatische Spracherkennung (Automatic Speech Recognition, ASR) hat sich in den letzten Jahrzehnten von regelbasierten Ansätzen hin zu datengetriebenen Methoden auf Basis neuronaler Netze entwickelt. Während klassische Systeme wie Hidden Markov Models (HMM) in Kombination mit Gaussian Mixture Models (GMM) lange Zeit den Stand der Technik darstellten, haben tiefe neuronale Netze (Deep Neural Networks, DNN) und insbesondere rekurrente sowie transformerbasierte Architekturen die Genauigkeit und Robustheit deutlich verbessert.

2.1 KOMMERZIELLE SYSTEME

Fabian Scherer

Im praktischen Einsatz dominieren aktuell cloudbasierte Spracherkennungsdienste großer Anbieter:

- **Google Speech-to-Text:** Bietet eine Echtzeit-Transkription in über 125 Sprachen und Dialekten. Das System nutzt Deep-Learning-Modelle, die kontinuierlich mit neuen Sprachdaten verbessert werden (Google Cloud, 2025).
- **Microsoft Azure Speech Services:** Ermöglicht neben Transkription auch Sprechertrennung („speaker diarization“) und individuelle Anpassung durch benutzerdefinierte Sprachmodelle (Microsoft, 2025).
- **Amazon Transcribe:** Konzentriert sich auf skalierbare Echtzeit-Transkription und bietet branchenspezifische Anpassungen, z. B. für Medizin oder Kundenservice (Amazon Web Services, 2025).

Diese Systeme sind leistungsfähig, erfordern jedoch meist eine Internetverbindung und bergen datenschutzrechtliche Herausforderungen, da Sprachdaten an externe Server übermittelt werden.

2.2 OPEN-SOURCE-ANSÄTZE

Fabian Scherer

Parallel dazu haben Open-Source-Lösungen an Bedeutung gewonnen. Besonders hervorzuheben ist Whisper von OpenAI, ein auf Transformer-Architekturen basierendes Modell, das in 2022 veröffentlicht wurde. Whisper zeichnet sich durch hohe Robustheit gegenüber Hintergrundgeräuschen und Akzenten aus und unterstützt Mehrsprachigkeit sowie Übersetzung. Da es lokal betrieben werden kann, eignet es sich für Szenarien mit hohen Datenschutzanforderungen.

Weitere Open-Source-Projekte wie Kaldi oder ESPnet haben sich in der Forschung etabliert und dienen als Grundlage für die Entwicklung und Evaluierung neuer Spracherkennungsalgorithmen.

Herausforderungen und Trends

Trotz der erheblichen Fortschritte bestehen weiterhin Herausforderungen. Dazu gehören die zuverlässige Sprechertrennung in Gruppensituationen, die Erkennung von Fachterminologie, sowie die Anpassung an individuelle Sprachgewohnheiten und Akzente. Zunehmend wird auch die Integration von selbstlernenden Systemen untersucht, die ihre Modelle durch fortlaufende Nutzung verbessern. Ein weiterer Trend ist die Echtzeit-Verarbeitung auf Edge-Geräten, die datenschutzfreundliche Lösungen ohne Cloud-Anbindung ermöglicht.

Für das vorliegende Projekt bedeutet dies, dass moderne KI-gestützte ASR-Systeme als technologische Basis genutzt werden können, während die gezielte Integration in Meeting- und Konferenzkontexte – einschließlich Datenbankanbindung, QR-Code-Verteilung und selbstlernender Komponenten – eine Weiterentwicklung des aktuellen Stands darstellt.

3

ÜBERBLICK ÜBER DIE APPLIKATION

Mike Wild

In diesem Kapitel wird die entwickelte Applikation vorgestellt. Ziel ist es, einen umfassenden Eindruck von den vorhandenen Funktionalitäten sowie der grafischen Benutzeroberfläche zu vermitteln. Dabei werden zunächst die zentralen Anwendungsbereiche und Hauptfunktionen erläutert. Anschließend erfolgt eine Darstellung der Benutzeroberfläche, um den Aufbau und die Bedienlogik der Software nachvollziehbar zu machen. Auf diese Weise wird ein zusammenhängendes Bild des Systems geschaffen, das sowohl die technischen Kernkomponenten als auch die praktische Nutzung aus Anwendersicht verdeutlicht.

3.1 ZIELSETZUNG UND EINSATZBEREICH

Mike Wild

Diese Applikation wurde entwickelt, um Online-Meetings zu protokollieren. Dabei ist es egal, mit welchem Programm das Meeting geführt wird oder auf welchem Betriebssystem, ob Windows, Linux oder MacOS. Ziel ist es die automatische Transkription in Echtzeit durchzuführen und die Transkripte in einer Datenbank zu speichern und zu verwalten. Das Programm bietet jedoch Potential, um auch in anderen Anwendungsbereichen eingesetzt zu werden (siehe dazu Kapitel 14, Seite 55). Die nachfolgend eingefügten Bilder der Benutzeroberfläche zeigen das Aussehen auf ein Linux-System mit systemseitig dunklen-Theme eingestellt.

3.2 FUNKTIONSUMFANG

Mike Wild

Um ein Meeting vollständig erfassen zu können, benötigt man neben dem Audiosignal des Mikrofons, was nur die eigene Stimme aufzeichnet, auch die Stimmen der anderen Teilnehmer. Deshalb greift die Applikation auch auf den Systemmix, das Ausgangs-Audiosignal, das an die Lautsprecher bzw. Kopfhörer geht, zu. Mikrofon-Audio und Systemmix werden gemischt und aufgezeichnet. Diese Aufzeichnung kann hinterher als Audiodatei gespeichert werden, um sich das Meeting oder Teile davon erneut anzuhören. Das erfasste Audiosignal wird entweder in Echtzeit oder nachgelagert von einem lokal-laufenden Künstliche Intelligenz (KI)-Modell transkribiert, wobei auch nach Sprecher getrennt wird. Mithilfe der Applikation ist es möglich das Transkript zu manipulieren, um Fehler der Transkription zu korrigieren oder den, von der KI erkannten Sprecher, echte Namen zuzuordnen. Das Transkript kann sowohl in der Datenbank, als auch als JSON-Datei gespeichert werden (siehe Abbildung 3.1). Des Weiteren ist ein Export als PDF-Datei möglich (vgl. Abbildung 3.2). Der Transkription können auch noch Schlagworte, sogenannte Tags, automatisch hinzugefügt werden.

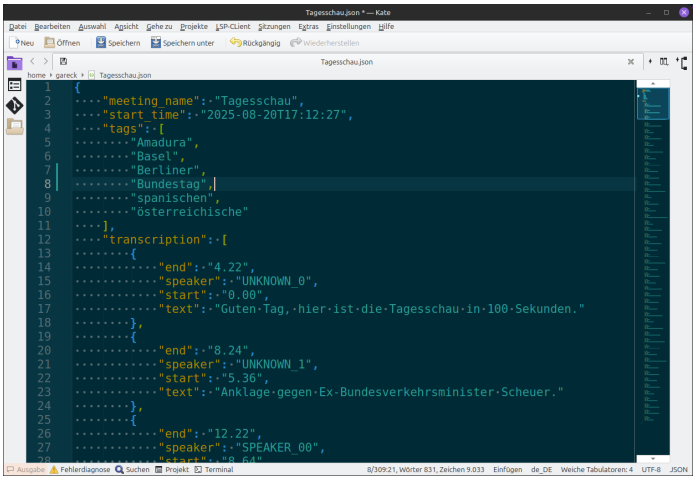

Abbildung 3.1: Ausschnitt einer exportierten JSON-Datei in einem Texteditor

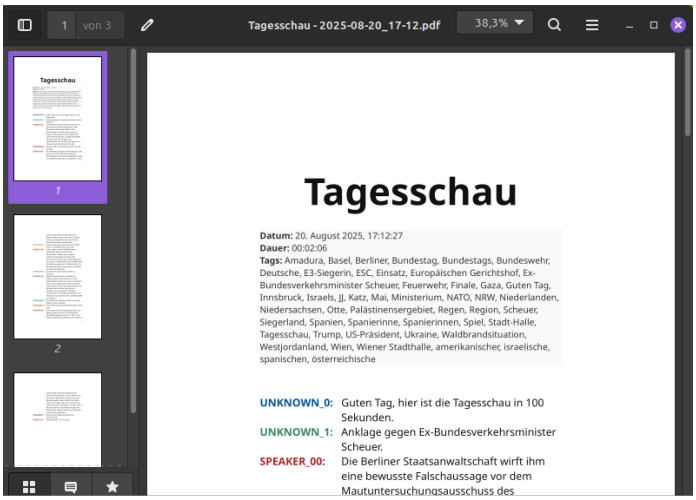

Abbildung 3.2: Anfang eines Transkriptes im PDF-Format

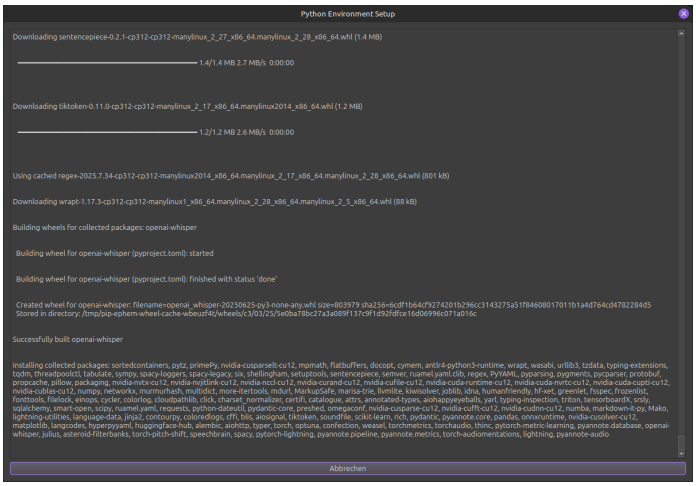

Abbildung 3.3: Python-Installationsfenster

3.3 ANLEITUNG UND BENUTZEROBERFLÄCHE

Mike Wild

3.3.1 *Python-Installationsfenster*

Wenn man das Programm das erste Mal startet, öffnet sich nicht das Hauptfenster der Anwendung, sondern es wird automatisch eine virtuelle Python-Umgebung eingerichtet, die alle benötigten Pakete und KI-Modelle für die Transkription und andere Funktionen beinhaltet. Dies dient dazu, dass das Programm das System, auf dem es läuft, nicht verändert. Abbildung 3.3 zeigt das Installationsfenster für die virtuelle Python-Umgebung. Die Installation übernimmt ein Shell- bzw. Batch-Skript. Die Ausgabe des Skripts wird im Fenster angezeigt. Das Skript kann später über den Menü-Punkt "Python neu-installierenöder auch manuell erneut gestartet werden. Die nun installierte virtuelle Umgebung ist nicht zwangsläufig notwendig. Man kann die Installation abbrechen und das auf dem System installierte Python nutzen. Hierzu wird nach dem Abbrechen oder bei einem Installationsfehler zunächst das Einstellungsfenster (siehe Abb. 3.4 geöffnet.

3.3.2 *Einstellungsfenster*

Mike Wild

Dort muss unter dem Punkt „Python-Pfad“ der Pfad zur ausführbaren Python-Datei (meist „python3“ genannt) eingetragen werden. Hier kann die Datei aus einer beliebigen virtuellen Umgebung oder die Datei vom System eingetragen werden. Unter dem zweiten Punkt „ASR-Skriptpfad“ muss ein Python-Skript zum Transkribieren eingetragen werden. Das Programm bietet momentan die beiden Möglichkeiten „run_asr.py“ und „asr_backend.py“. Das erste Skript transkribiert das Audio erst hinterher, indem es eine Wav-Datei entgegennimmt. Es arbeitet mit pyannote und whisper. Das zweite Skript ist für die Echtzeitverarbeitung zuständig, näheres im Kapitel 11 auf Seite 36. Diese ersten zwei Punkte sind essentiell, damit das Programm wie vorgesehen arbeiten kann.

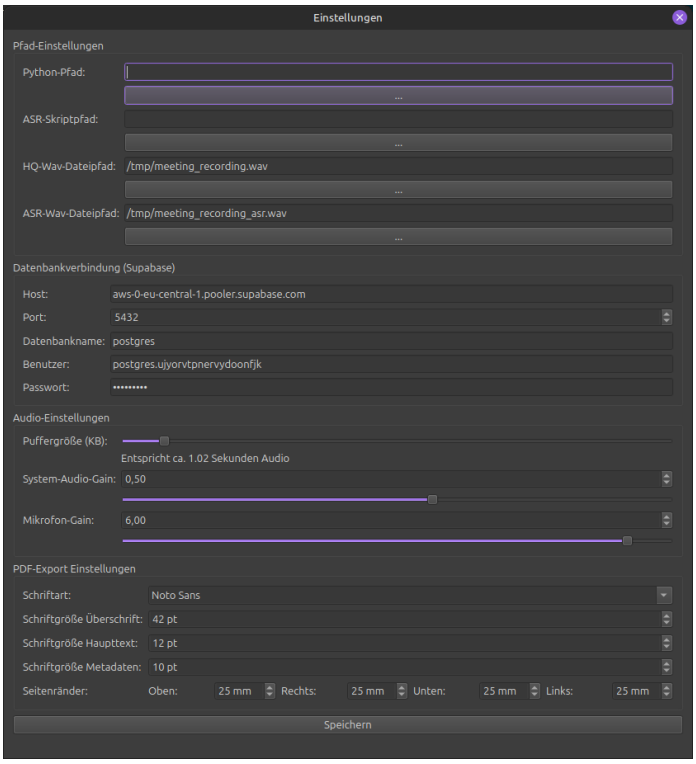

Abbildung 3.4: Einstellungsfenster

Mit den nächsten zwei Pfaden kann man angeben, wo das Programm die Audiodateien zwischenspeichert. Diese werden im Wav-Format gespeichert. Dabei steht „HQ“ für **H**igh-**Q**uality, also die Audiodatei mit voller Qualität, und „ASR“ für die downgesamplete Audiodatei zur nachträglichen Transkription. Standardmäßig werden für die diese beiden Dateien in einem Standardordner für temporäre Dateien gespeichert. Danach folgen Einstellungen zur Datenbankverbindung, wo die Transkripte gespeichert und abgerufen werden sollen. Hier ist eine „Supabase“ Datenbank vorgesehen, kann aber ggf. durch eine andere Datenbank ausgetauscht werden, die im Qt-Framework vom Typ „QPSQL“ ist und mit denselben Parametern sich verbinden lässt. Anschließend folgend die Audio-Einstellungen. Die Puffergröße bestimmt wie viel Audio-Daten zwischengespeichert werden, bevor sie auf die Festplatte in die Wav-Dateien geschrieben werden. Die beiden Gain-Werte ergeben das Mischungsverhältnis zwischen Mikrofon- und Systemaudio-Lautstärke. Diese sollten so eingestellt werden, dass in der resultierenden Audio-Datei die Lautstärke des Mikrofons genauso laut ist, wie die des Systemmix. Zuletzt folgen noch Einstellungen, um das Aussehen der PDF-Datei, die man aus dem Transkript erzeugen lassen kann, zu optimieren.

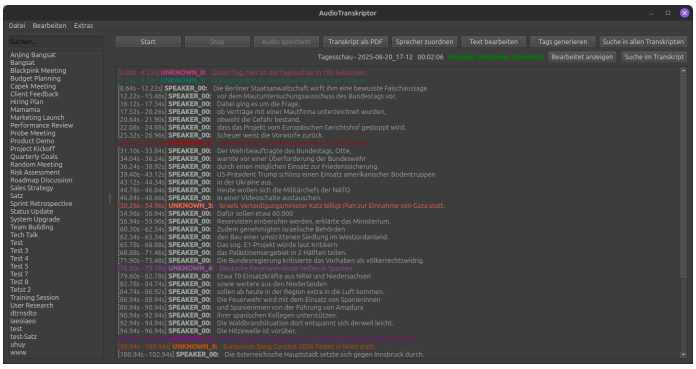

Abbildung 3.5: Hauptfenster der Applikation

3.3.3 *Hauptfenster*

Mike Wild

Im Hauptfenster der Applikation befindet sich ganz oben die Menü-Leiste mit der man Transkripte speichern und laden kann, Änderungen rückgängig machen, Python neu installieren kann und das Einstellungsfenster öffnet. Bei MacOS befindet sich die Menüleiste nicht im Fenster, sondern am oberen Bildschirmrand. Auf der linken Seite sieht man die in der Datenbank gespeicherten Transkripte. Auf der rechten Seite sieht man zentral das gerade geladene Transkript und oberhalb davon die Bedienelemente. „Start“ bzw. „Stopp“ starten bzw. stoppen die Audioaufnahme. Erst nachdem eine Aufnahme beendet worden ist, kann man mit „Audio speichern“ diese als Wav-Datei speichern. „Transkript als PDF“ dient zum Exportieren des Transkriptes als PDF-Datei. „Sprecher zuordnen“ (siehe Abschnitt 3.3.4) und „Text bearbeiten“ (siehe Abschnitt 3.3.5) dienen zum Manipulieren des Transkriptes. Mit „Tags generieren“ werden über ein Python-Skript mittes spaCy Schlagworte zum gesamten Transkript erzeugt.

Yolanda Hadiana Fiska

Der Button „Bearbeitet anzeigen“ ermöglicht es dem Benutzer, die Ansicht des geladenen Transkripts abhängig vom aktuellen Modus umzuschalten. Während die Option „Bearbeitet anzeigen“ zur redigierten Version des Transkripts wechselt, führt die Option „Original anzeigen“ zurück zur ursprünglichen Fassung. Der jeweils aktive Modus wird links neben diesem Umschaltknopf angezeigt. Darüber hinaus stehen dem Benutzer zwei Funktionen zur Protokollsuche zur Verfügung: Mit „Suche im Transkript“ kann innerhalb des aktuell geladenen Transkripts nach bestimmten Begriffen, Diskussionspunkten oder Informationen gesucht werden (siehe Abschnitt 12.4.2.4), während „Suche in allen Transkripten“ eine suchübergreifende Recherche über sämtliche vorhandenen Transkripte hinweg ermöglicht (siehe Abschnitt 12.4.2.4).

3.3.4 *Sprecherbearbeitung*

Mike Wild

Das Fenster zum Bearbeiten bzw. Zuordnen von Sprecher hat zwei Tabs, zwischen denen man wechseln kann. Der erste Tab (Abbildung 3.6) dient dazu den Sprechern im Transkript, die automatisch mit „SPEAKER_XX“

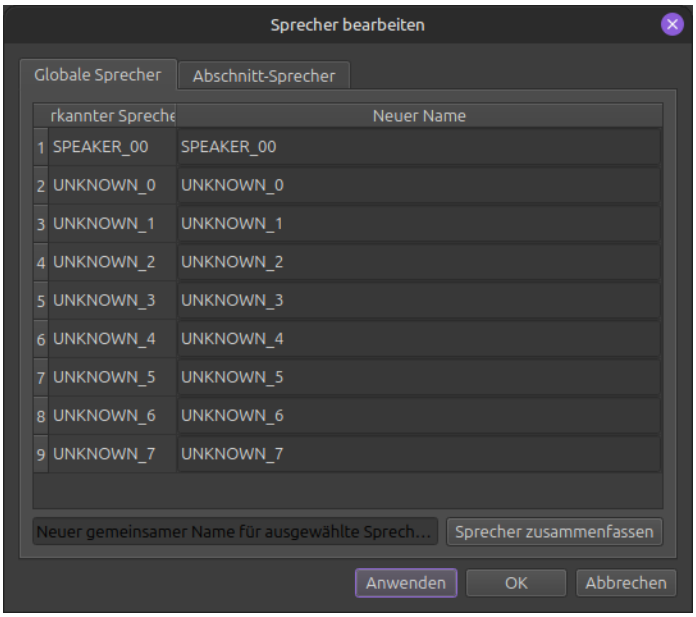

Abbildung 3.6: Sprecherzuordnung für das gesamte Transkript

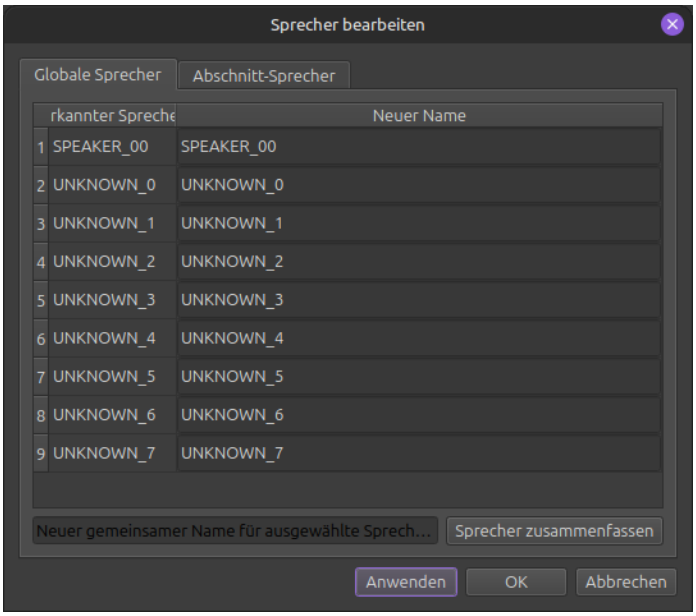

Abbildung 3.7: Sprecherzuordnung für einzelne Abschnitte

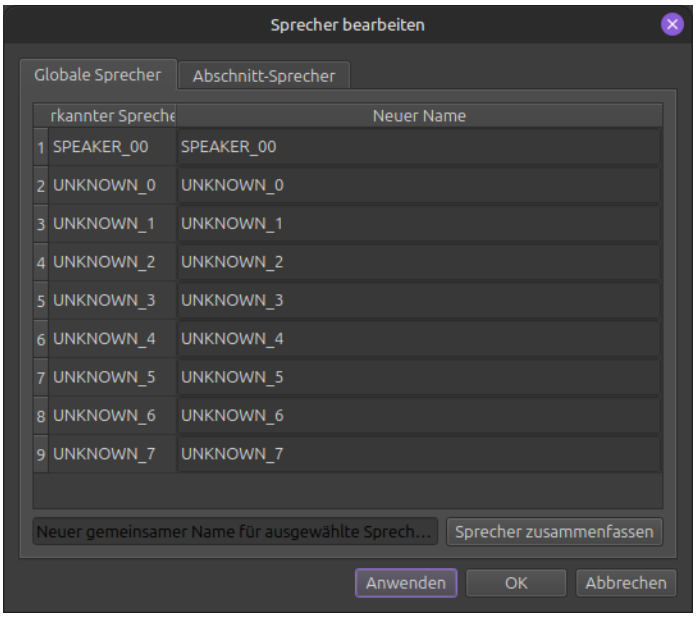

Abbildung 3.8: Textbearbeitung einzelner Abschnitte

(wobei XX eine zweistellige Nummer ist) benannt sind, echte Namen zuzuordnen. Das heißt, der Sprechername wird im gesamten Transkript ausgetauscht. Der zweite Tab (Abbildung 3.7) ist dafür gedacht, falls die KI den Text einen falschen Sprecher zuordnet, dies für diesen Textabschnitt zu korrigieren. Hier lassen sich die Spalten Start(-zeit) Ende (Endzeit) und Text nicht bearbeiten und dienen zum eindeutigen Erkennen des Abschnittes, von den man den Sprecher ändern möchte.

3.3.5 *Textbearbeitung*

Mike Wild

Mit dem Fenster in Abbildung 3.8 kann man den von der KI automatisch transkribierten Text korrigieren, falls diese etwas fehlerhaft transkribiert hat. Die Spalten Start(-zeit) Ende (Endzeit) und Sprecher lassen sich hier nicht bearbeiten und dienen zum eindeutigen Erkennen des Abschnittes, den man bearbeiten möchte.

3.3.6 *Suche innerhalb eines Meetings*

Yolanda Hadiana Fiska

Das Fenster „Erweiterte Suche im Transkript“ (siehe Abbildung 3.9) erlaubt eine gezielte Recherche innerhalb des aktuell geöffneten Transkripts. Es stellt verschiedene Such- und Filteroptionen zur Verfügung. Über das Feld „Schlüsselwort“ kann nach einem oder mehreren spezifischen Begriffen gesucht werden. Zusätzlich besteht die Möglichkeit, die Suche nach Sprecher (z. B. alle Sprecher oder ein bestimmter Redner) sowie nach Tags einzugrenzen. Ebenso können durch die Angabe einer Startzeit und Endzeit nur bestimmte Zeitbereiche des Transkripts berücksichtigt werden.

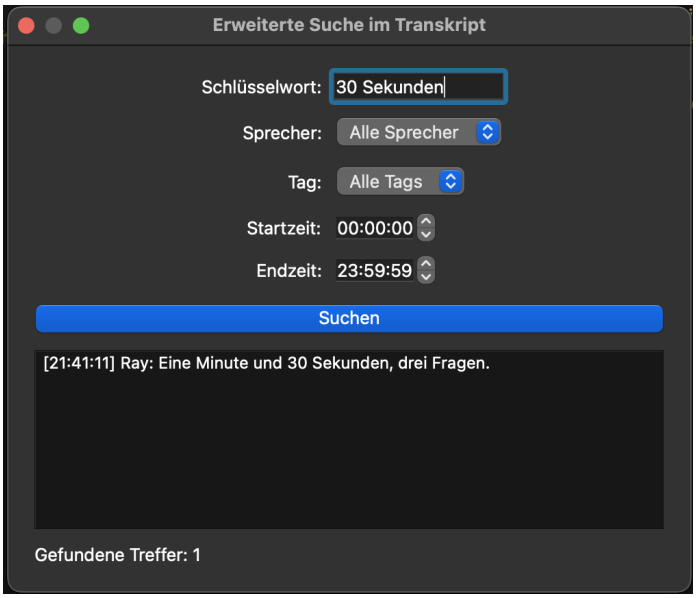

Abbildung 3.9: Systemansicht: Suche im Einzeltranskript

Die Ergebnisse der Suche werden in einer Liste dargestellt und enthalten die Zeitangabe, den Namen des Sprechers sowie die entsprechende Aussage. Wie in Abbildung 3.9 dargestellt, werden die Eingabeoptionen für die Suche bereitgestellt.

3.3.7 *Globale Suche über alle Meetings*

Yolanda Hadiana Fiska

Das Fenster „Erweiterte Mehrfachsuche“ (siehe Abbildung 3.10) ermöglicht eine protokollübergreifende Recherche über sämtliche gespeicherten Transkripte. Im Unterschied zur Suche innerhalb eines einzelnen Protokolls werden hier alle verfügbaren Sitzungen in die Abfrage einbezogen. Die Filtermöglichkeiten entsprechen im Wesentlichen denen der Suche innerhalb eines Meetings, werden jedoch um zusätzliche Optionen wie die Eingrenzung nach Datum oder nach einem bestimmten Zeitraum erweitert.

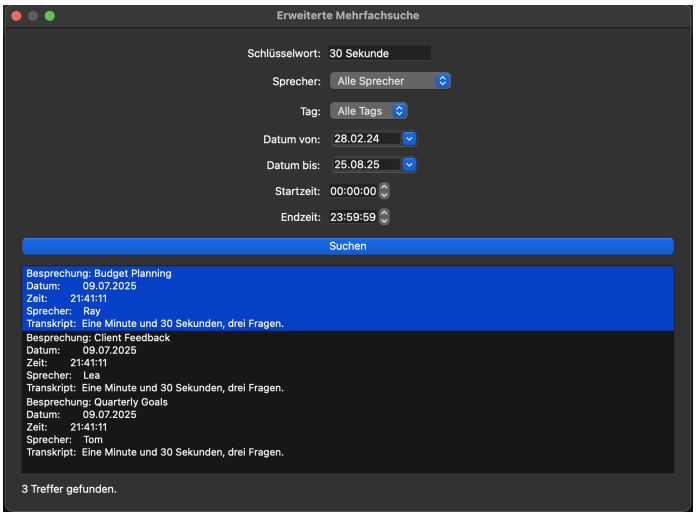

Abbildung 3.10: Systemansicht: Suche über alle Transkripte

Die Suchergebnisse werden in einer zentralen Liste angezeigt und enthalten neben dem Namen der Sitzung, dem Datum und dem Sprecher auch den Ausschnitt der Aussage, in dem das gesuchte Schlüsselwort vorkommt. Wie in Abbildung 3.10 dargestellt, kann durch Anklicken eines Eintrags direkt das entsprechende Meeting im Hauptfenster geladen werden. Dort werden die gesuchten Begriffe im Transkript hervorgehoben, sodass die relevanten Stellen unmittelbar erkennbar sind (Siehe Abbildung 3.11).

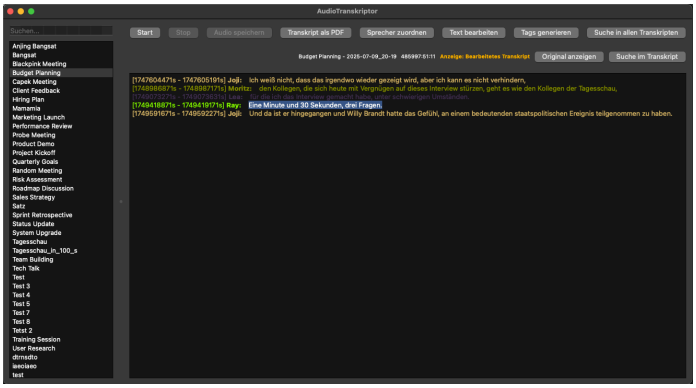

Abbildung 3.11: Darstellung eines Transkripts mit hervorgehobenen Suchergebnissen nach Auswahl eines Eintrags aus der Ergebnisliste

3.4 WORKFLOW AUS ANWENDERSICHT

Mike Wild

Angenommen man hat das Programm schon einmal gestartet, sodass die Python-Umgebung richtig aufgesetzt ist, dann sieht der typische Workflow mit dieser Applikation wie folgt aus. Das Programm startet mit dem Haupt-

fenster (vgl. Abschnitt 3.3.3, Seite 8). Man drückt auf „Start“, hält sein Online-Meeting ab und drückt auf „Stop“. Danach kann man die Audiodatei speichern. Anschließend ordnet man den Sprecher die entsprechenden Namen zu, korrigiert ggf. den Text, generiert Schlagworte für das Transkript und speichert es in der Datenbank. Bei Bedarf exportiert man das Transkript als PDF-Datei. Danach ist man bereit für das nächste Meeting.

4

KONZEPTION & ARCHITEKTUR DER ANWENDUNG

Mike Wild

Dieses Kapitel beschreibt die fachliche Konzeption und die Softwarearchitektur der Anwendung. Ziel ist eine plattformübergreifende, echtzeitfähige Erfassung von System- und Mikrofon-Audio, deren automatische Transkription und Weiterverarbeitung und Verwaltung des Textes.Der Fokus dieses Kapitels liegt auf den architektonischen Entscheidungen, die getroffen wurden, um die zentralen Herausforderungen zu bewältigen. Es werden die Schlüsselkomponenten vorgestellt und die zugrundeliegenden Design Patterns erläutert.

4.1 ARCHITEKTONISCHER ÜBERBLICK UND DESIGN-PRINZIPIEN

Mike Wild

Die Software-Architektur der Applikation basiert auf den **„Single Responsibility Prinzip“, „Open-Closed Prinzip“, „Liskovsches Substitutionsprinzip“, „Interface Segregation Prinzip“, „Dependency Inversion Prinzip“ (SOLID)**-Prinzipien der objektorientierten Programmierung.

“Das **Single-Responsibility-Prinzip** besagt, dass jede Klasse nur eine einzige Verantwortung haben solle. Verantwortung wird hierbei als „Grund zur Änderung“ definiert. [...] Das **Open-Closed-Prinzip** besagt, dass Software-Einheiten (hier Module, Klassen, Methoden usw.) Erweiterungen möglich machen sollen (dafür offen sein), aber ohne dabei ihr Verhalten zu ändern (ihr Sourcecode und ihre Schnittstelle sollte sich nicht ändern). [...] Das **Liskovsche Substitutionsprinzip (LSP)** oder Ersetzbarkeitsprinzip fordert, dass eine Instanz einer abgeleiteten Klasse sich so verhalten muss, dass jemand, der meint, ein Objekt der Basisklasse vor sich zu haben, nicht durch unerwartetes Verhalten überrascht wird, wenn es sich dabei tatsächlich um ein Objekt eines Subtyps handelt. [...] Das **Interface-Segregation-Prinzip** dient dazu, zu große Interfaces aufzuteilen. Die Aufteilung soll gemäß den Anforderungen der Clients an die Interfaces gemacht werden – und zwar derart, dass die neuen Interfaces genau auf die Anforderungen der einzelnen Clients passen. Die Clients müssen also nur mit Interfaces agieren, die das und nur das können, was die Clients benötigen. [...] Das **Dependency-Inversion-Prinzip** beschäftigt sich mit der Reduktion der Kopplung von Modulen. Es besagt, dass Abhängigkeiten immer von konkreteren Modulen niedriger Ebenen zu abstrakten Modulen höherer Ebenen gerichtet sein sollten. “ [7]

Von der Architektur her wird auf die Trennung von Verantwortlichkeiten geachtet. So ist die Benutzeroberfläche (`MainWindow`) von der Backend-Logik entkoppelt. Aufwendige Operationen, wie die Audio-Aufnhame, das Schreiben von Dateien und die Kommunikation mit externen Python-Prozessen werden in separate Hintergrund-Threads und spezialisierte Manager-Klassen

ausgelagert. Dieser Ansatz stellt sicher, dass die grafische Benutzeroberfläche (**G**raphical **U**ser **I**nterface) (GUI) jederzeit reaktionsfähig bleibt.

4.2 HAUPTKOMPONENTEN

Mike Wild

Hier wird ein kurzer Überblick über die wichtigsten Komponenten der Anwendung gezeigt. Diese sind:

- **`MainWindow`**: Agiert als zentrale **Controller**-Instanz. Sie initialisiert alle Sub-Systeme, stellt die GUI dar und reagiert auf Benutzerinteraktionen, indem sie Aufgaben an die zuständigen Manager-Klassen delegiert.
- **Manager-Klassen (`FileManager`, `AsrProcessManager`, etc.)**: Kapseln spezifische Logikbereiche. Sie steuern externe Prozesse und abstrahieren den Dateizugriff.
- **`Transcription`**: Dient als zentrales **Datenmodell (Model)**, das alle Informationen zu einem Meeting enthält und als alleinige Quelle der Wahrheit („Single Source of Truth“) fungiert.
- **`AudioFactory`**: Erzeugt plattformabhängig das richtige Audio-Aufnahme-Objekt und gibt es als `CaptureThread` zurück.
- **`CaptureThread`**: Ist eine abstrakte Basisklasse, die die plattformabhängigen Audio-Aufnahmen kapselt und gleichzeitig mittels `QWaitCondition` dafür sorgt, dass Ressourcen gespart werden, indem der Thread nur dann arbeitet, wenn auch Daten vorhanden sind und ansonsten schläft.
- **Audio-Threads (`CaptureThread`, `WavWriterThread`)**: Implementieren das **Producer-Consumer-Pattern**. Der `CaptureThread` produziert Audiodaten, während der `WavWriterThread` diese konsumiert und auf die Festplatte schreibt.

4.3 THREADING- UND SYNCHRONISATIONSKONZEPT

Mike Wild

Die Audioaufnahme und das Dateischreiben laufen in eigenen Threads. Der GUI-Thread bearbeitet ausschließlich Benutzerinteraktionen und reagiert über Signale/Slots:

- **CaptureThread** (Audio): nicht-blockierendes Einlesen; Datenübergabe via `emit pcmChunkReady(...)` an den `WavWriterThread` mit `Qt::QueuedConnection`.
- **WavWriterThread**: wartet auf Daten, schreibt blockweise, finalisiert Header.
- **GUI-Thread**: steuert mit Start/Stop; aktualisiert Anzeigen (Zeitlabel / Status); lässt bei `Audio speichern` die HQ-Wav-Datei via `QtConcurrent::run` kopieren; verwaltet externe Prozesse via `QProcess`.

Das Beenden wird explizit orchestriert: `stopCapture()` beendet den inneren Loop, das `stopped`-Signal initiiert `stopWriting()`, anschließend löst `finishedWriting` den Automatic Speech Recognition = Automatische Spracherkennung (ASR)-Anstoß aus. `shutdown()`-Methoden stellen einen sauberen Thread-Stopp mit `wait()` sicher.

4.4 FEHLERBEHANDLUNG UND CLEANUP

Mike Wild

Fehler im Initialisieren von Audio-Backends führen zu einem definierten Abbruch mit Aufräumen (`PulseAudio`-Module entladen; `WASAPI`-Interfaces *safe release*). Beim Beenden der Anwendung werden Aufnahme- und Writer-Thread via `shutdown()` synchron gestoppt. Der `AsrProcessManager` signalisiert Erfolgs-/Fehlerzustände an die GUI; Fehlermeldungen aus *stderr* werden an die Nutzer weitergereicht.

4.5 KONFIGURIERBARKEIT

Mike Wild

Zentrale Parameter werden über `QSettings` verwaltet: Lautstärkegewichte (`sysGain`, `micGain`), Puffergröße, Größe und Position des Fensters, Pfade der Python-Umgebung und ASR-Skripte. Damit sind Build- und Laufzeitbelange getrennt und reproduzierbar dokumentiert.

4.6 ZUSAMMENFASSUNG

Mike Wild

Die Architektur separiert Aufgaben konsequent: Audioaufnahme (plattformnah) in spezialisierten Threads, persistentes Schreiben als eigenständige Stufe, ASR als externer Prozess sowie eine GUI, die lediglich orchestriert. Die lose Kopplung über Signale/Slots und Prozesse schafft Robustheit, Testbarkeit und Erweiterbarkeit.

5 IMPLEMENTIERUNG DER SCHLÜSSELKOMPONENTEN

Mike Wild

Aufbauend auf dem in Kapitel 4 vorgestellten Konzept, erläutert dieses Kapitel die konkrete technische Umsetzung der zentralen Komponenten der Applikation. Der Schwerpunkt liegt auf der Realisierung der plattformübergreifenden Audio-Aufnahme, der Synchronisation, der Audio-Streams und der Anbindung der externen KI-Module.

5.1 DIE PLATTFORMÜBERGREIFENDE AUDIO-AUFNAHME

Mike Wild

Die größte Herausforderung bei der Audio-Aufnahme war die Inkompatibilität der nativen Audio-Schnittstellen der Zielplattformen MacOS, Windows (WASAPI) und Linux (PulseAudio). Um eine wartbare und erweiterbare Lösung zu schaffen, wurde eine Abstraktionsschicht implementiert, die auf zwei etablierten Software-Entwurfsmustern basiert.

5.1.1 *Abstraktion durch Design Patterns*

Mike Wild

Die Entkopplung der plattformspezifischen Logik von der Hauptanwendung wurde durch die kombinierte Anwendung des *Abstract Factory*- und des *Template Method*-Patterns erreicht.

- **Template Method Pattern:** Die abstrakte Basisklasse `CaptureThread` definiert das Grundgerüst des Aufnahme-Algorithmus in ihrer `run()`-Methode. Dieser Ablauf – Initialisieren, Aufnahmeschleife, Aufräumen – ist für alle Plattformen identisch. Die konkreten, plattformabhängigen Implementierungsschritte werden als rein virtuelle Methoden (`initializeCapture()`, `captureLoopIteration()`, `cleanupCapture()`) deklariert, die von den abgeleiteten Klassen überschrieben werden müssen.
- **Abstract Factory Pattern:** Die `AudioFactory` dient als zentrale Erzeugungsinstanz. Ihre statische Methode `createThread()` wählt zur Kompilierzeit mittels Präprozessor-Direktiven (`#if defined(Q_OS_WIN)`) die korrekte, konkrete `CaptureThread`-Implementierung aus und instanziiert diese. Dadurch bleibt der restliche Code, insbesondere die `MainWindow`, vollständig agnostisch gegenüber dem zugrundeliegenden Betriebssystem.

Wie die Aufnahme der Audiosignale konkret implementiert ist, kann man in den nachfolgenden Kapiteln sehen. Die Audioaufnahme unter Windows in Kapitel 6 ab Seite 19, für Linux (nur PulseAudio) in Kapitel 7 ab Seite 21 und für MacOS in Kapitel 8 ab Seite 24.

5.2 ANBINDUNG DER KI-MODULE

Mike Wild

Die Ausführung der rechenintensiven Python-Skripte für die Transkription und Tag-Erstellung ist in den Manager-Klassen `AsrProcessManager` und `TagGeneratorManager` gekapselt. Beide folgen demselben Entwurfsmuster, um die asynchrone und sichere Kommunikation zwischen C++ und Python zu gewährleisten.

Der Manager startet das jeweilige Python-Skript als externen Prozess mittels `QProcess`. Die Datenübergabe von C++ nach Python erfolgt über die Standard-Eingabe (stdin) des Prozesses. Dies ist besonders für die Tag-Generierung vorteilhaft, da so beliebig lange Texte ohne Beschränkungen durch Kommandozeilenargumente übergeben werden können. Nach dem Schreiben der Daten wird der Schreibkanal mit `closeWriteChannel()` geschlossen, was dem Python-Skript das Dateiende (EOF) signalisiert und dessen Verarbeitung anstößt.

Die Ergebnisse werden vom Python-Skript zeilenweise auf die Standard-Ausgabe (stdout) geschrieben. Der C++-Manager liest diese Ausgabe, parst sie und kommuniziert die fertigen Ergebnisse (z.B. ein `MetaText`-Segment oder eine Liste von Tags) über Qt-Signale an die `MainWindow`. Dieser ereignisbasierte Ansatz stellt sicher, dass die GUI auch während der Analyse durch die Python-Skripte nicht blockiert wird.

5.3 NACHGELAGERTE TRANSKRIPTION

Mike Wild

Die Aufgabenstellung hat eine Transkription in Echtzeit gefordert, dennoch ist in der Applikation auch noch eine nachgelagerte Transkription mit eingebaut. Dies hat zweierlei Gründe. Zum einen würde Übergangsweise eine Lösung benötigt bis das eigentliche KI-Modell fertig war, um die Applikation vollständig zu Testen und die anderen Features entwickeln zu können. Und zum anderen ist das eine gute Möglichkeit die Ergebnisse der Echtzeittranskription zu vergleichen und zu evaluieren. Das hier eingesetzte Python-Skript (`run_asr.py`) verwendet `pyannote` für die Sprecherdiarisierung und `whisper` für die Transkription. Dies lässt sich aber leicht ändern, sodass beispielsweise die Transkription über `google` erfolgen kann. Da die nachgelagerte Transkription nicht Teil der Aufgabenstellung ist, wird hier auch nicht näher darauf eingegangen. Somit beschäftigt sich Kapitel 11 mit der echtzeitfähigen Transkription.

HINWEIS

Aufgrund des Zeitmangels sind die Komponenten: „Audioaufnahme unter MacOS“ und „Echtzeittranskription“ noch nicht vollständig in der endgültigen Version der Applikation (main-Branch) enthalten. Diese sind jedoch einzeln in einer eigenen Version integriert (siehe Paki-Branch, bzw. Fabian-Branch).

6 AUDIOAUFNAHME UNTER WINDOWS (WASAPI)

Mike Wild

Unter Windows wird die Audioaufnahme über die Windows Audio Session API (WASAPI) realisiert. Die Klasse `WinCaptureThread` kombiniert *Loopback*-Capture für den Systemmix (Wiedergabe) und reguläres Capture für das Mikrofon (Aufnahme). Die Implementierung adressiert zwei zentrale Herausforderungen: (I) Synchronisation zweier unabhängiger Geräte mit potentiell unterschiedlicher Samplerate und (II) robuste Pufferung/Entkopplung durch ringförmige FIFOs und einen zeitbasierten Resampling-Ansatz.

6.1 DESIGNZIELE UND GRUNDIDEE

Mike Wild

- **Gleichzeitiges Erfassen** von System- und Mikrofon-Stream via getrennte `IAudioClient/IAudioCaptureClient`-Paare.
- **Zeitbasierte Synchronisation**: statt paketgetriebener Taktung wird die exakte vergangene Zeit per `QueryPerformanceCounter` ermittelt; daraus wird die zu generierende Zielframezahl bei 48 kHz abgeleitet.
- **Sanftes Resampling**: FIFO-Puffer + kontinuierliche Lese-Positionen (`m_resampPosSys/Mic`) erlauben interpolationsfreies (nearest/linear) Sampling; hier: Sample-At-Position via Ringpuffer.
- **Thread-Lebenszyklus** durch Basisklasse `CaptureThread` (Warten → Initialisieren → Loop → Cleanup).

6.2 INITIALISIERUNG (`initializecapture()`)

Mike Wild

Ablaufbeschreibung

1. **Geräte-Enumerator**: `MMDeviceEnumerator` erzeugen.
2. **Default-Geräte**: Wiedergabe (`eRender`, `eConsole`) für Loopback, Aufnahme (`eCapture`, `eConsole`) für Mikrofon.
3. **IAudioClient**: je Gerät aktivieren und `GetMixFormat()` abfragen (native Samplerate/Kanalzahl).
4. **Initialize()**: Systemclient im Shared-Mode mit `AUDCLNT_STREAMFLAGS_LOOPBACK`; Mikrofonclient im Shared-Mode.
5. **IAudioCaptureClient**: je Client abrufen; Puffergrößen vorbereiten.
6. **Start()**: beide Streams starten.

Auszug (Listing)

Listing 6.1: WASAPI-Setup (vereinfacht)

```
hr = CoCreateInstance(__uuidof(MMDeviceEnumerator), nullptr,
    CLSCTX_ALL,
IID_PPV_ARGS(&m_deviceEnumerator));
m_deviceEnumerator->GetDefaultAudioEndpoint(eRender, eConsole, &
    m_deviceSys);
m_deviceSys->Activate(__uuidof(IAudioClient), CLSCTX_ALL, nullptr,
reinterpret_cast<void**>(&m_audioClientSys));
WAVEFORMATEX* wfexSys = nullptr;
m_audioClientSys->GetMixFormat(&wfexSys);
m_audioClientSys->Initialize(AUDCLNT_SHAREMODE_SHARED,
AUDCLNT_STREAMFLAGS_LOOPBACK,
10000000, 0, wfexSys, nullptr);
CoTaskMemFree(wfexSys);
m_audioClientSys->GetService(IID_PPV_ARGS(&m_captureClientSys));
// ... analog fuer Mic (ohne LOOPBACK-Flag)
```

6.3 ZEITBASIERTE SYNCHRONISATION & RESAMPLING (captureloopiteration())

Mike Wild

Die Besonderheit bei der Audioaufnahme unter Windows mittels Windows Audio Session API (WASAPI) ist, dass man diese nicht paketgetrieben *zu takten* kann. Denn dann würden nur dann Daten aufgezeichnet werden, wenn auch tatsächlich Audiodaten zur Verfügung stehen, d.h. es wird keine Stille aufgezeichnet. Dies führt neben der falschen Aufnahme-Dauer auch noch zu dem Problem, dass die Audiosignale vom System und vom Mikrofon nicht mehr synchron währen. Deswegen muss, mithilfe eines sehr genauen Timers, die vergangene Zeit seit dem letzten Verarbeiten von Audiosignalen gemessen werden, und die Audiodaten um die Stille ergänzt werden. Somit wird die in der aktuellen Iteration vergangene Zeit Δt per `QueryPerformanceCounter` bestimmt. Daraus ergeben sich die zu erzeugenden Ziel-Frames bei 48 kHz:

$$N_{\text{target}} := \lfloor 48000 \cdot \Delta t + \text{Akkumulator} \rfloor.$$

Für jedes Zielsample werden die momentanen Werte aus den FIFO-Puffern mit Positionszeigern `m_resampPosSys/Mic` gelesen. Dieses Schema erzeugt bei ausbleibenden Paketen faktisch Stille, hält aber die Streams zueinander synchron.

7

AUDIOAUFNAHME UNTER LINUX (PULSEAUDIO)

Mike Wild

Dieses Kapitel beschreibt die plattformspezifische Audioaufnahme unter Linux mittels *PulseAudio*. Andere Audio-Systeme werden derzeit nicht unterstützt. Die Klasse `PulseCaptureThread` leitet von der abstrakten Basisklasse `CaptureThread` ab und implementiert die Initialisierung, den Aufnahme-Loop sowie das Aufräumen der PulseAudio-Ressourcen. Kernidee ist die gleichzeitige Erfassung des *Systemmixes* (Wiedergabegerät) und des *Mikrofons*, deren Signale in 32-bit-Float gemischt, gegen Clipping begrenzt und anschließend an den `WavWriterThread` übertragen werden.

7.1 ZIELE UND DESIGNENTSCHEIDUNGEN

Mike Wild

Die Linux-Implementierung verfolgt folgende Ziele:

- **Gleichzeitige Erfassung** von System-Audio (“what-you-hear”) und Mikrofon.
- **Robustheit** durch explizites Fehler-Handling beim Laden desModuls und beim Öffnen der PulseAudio-Streams.
- **Einfache Parametrisierung** (z. B. Lautstärkeverhältnisse) via `QSettings`.
- **Streckenweise Entkopplung** durch den `CaptureThread`-Lebenszyklus (Warteschleife → Initialisierung → Aufnahme → Cleanup).

7.2 ARCHITEKTUR UND DATENPFAD

Mike Wild

Die Initialisierung richtet zwei Aufnahmequellen ein:

1. **Systemmix** via `$DefaultSink.monitor`.
2. **Mikrofon** via *virtuelle Senke*: `module-null-sink` (“MicSink”) und ein `module-loopback` vom Standard-*Source* auf diese Senke; abgehört wird `mic_sink.monitor`.

Beide Quellen werden mit `pa_simple_new` als Float32LE, 48 kHz, Stereo geöffnet. Im Aufnahmeloop werden Blockweise $N = 1024$ Frames pro Quelle gelesen, in einem Mischpuffer skaliert (`sysGain`, `micGain`) und mit `qBound` gegen Clipping begrenzt. Der gemischte Block wird als `QList<float>` per `emit pcmChunkReady(chunk)` an den `WavWriterThread` übergeben (vgl. Kapitel 9). Anders als bei der Audioaufnahme unter Windows (siehe Abschnitt 6.3), kann hier auf ein Timer verzichtet werden, da `PulseAudio` auch bei Stille Audiodaten liefert.

7.3 INITIALISIERUNGSSCHRITTE (initializecapture())

Mike Wild

Ablauf in Worten

1. **Einstellungen laden**: sysGain, micGain aus QSettings.
2. **Standardgeräte ermitteln**: pactl info parsen (Sprache robust: EN/-DE).
3. **Virtuelle Module laden**: module-null-sink → mic_sink, module-loopback (Mikrofon → mic_sink).
4. **PulseAudio-Streams öffnen**: pa_simple_new für $DefaultSink.monitor (Systemmix) und mic_sink.monitor.
5. **Puffer vorbereiten**: bufSys, bufMic, bufMix mit $N \times 2$ (Stereo).

Auszug (Listing)

Listing 7.1: Ermitteln der Default-Geräte und Laden der PulseAudio-Module

```
QProcess infoProc;
infoProc.start("pactl", {"info"});
infoProc.waitForFinished(500);
QString info = QString::fromUtf8(infoProc.readAllStandardOutput());

QRegularExpression reEnSink(R"(Default Sink:\s*(\S+))"),
reDeSink(R"(Standard-Ziel:\s*(\S+))"),
reEnSrc (R"(Default Source:\s*(\S+))"),
reDeSrc (R"(Standard-Quelle:\s*(\S+))");

QString origSink = ...; // via RegEx
QString origSource = ...;

auto loadModule = [&](const QString& params)->int {
    QProcess p; p.start("pactl", QStringList{"load-module"} <<
        params.split(' '));
    p.waitForFinished();
    bool ok; int id = QString::fromUtf8(p.readAllStandardOutput()).
        trimmed().toInt(&ok);
    return ok ? id : -1;
};

m_modNull = loadModule("module-null-sink sink_name=mic_sink
    sink_properties=device.description=MicSink");
m_modLoop = loadModule(QString("module-loopback source=%1 sink=
    mic_sink").arg(origSource));
```

7.4 AUFNAHME-LOOP (captureloopiteration())

Mike Wild

Ablauf in Worten

In jeder Iteration werden $N = 1024$ Frames pro Quelle synchron gelesen. Schlägt `pa_simple_read` fehl, wird die Session beendet (`stopCapture()`). War das Lesen erfolgreich, werden anschließend System- und Mikrofonsamples gewichtet addiert:

$$\mathrm{mix}[i] = \mathrm{qBound}\big(-1,\ \mathrm{sysGain} \cdot \mathrm{sys}[i] + \mathrm{micGain} \cdot \mathrm{mic}[i],\ 1\big).$$

Der resultierende Stereo-Block `bufMix` wird in eine `QList<float>` kopiert, als `chunk` emittiert und vom `WavWriterThread` asynchron weiterverarbeitet bzw. resamplet und ans Echtzeit-Transkriptions-Skript weitergereicht.

7.5 AUFRÄUMEN (cleanupcapture())

Mike Wild

Beim Beenden werden offene Restpuffer optional geleert (“drain”) und beide `pa_simple`-Streams freigegeben. Anschließend entlädt die Implementierung die geladenen PulseAudio-Module (`module-loopback`, `module-null-sink`), um das System in den Ausgangszustand zurückzuversetzen. Dies ist insbesondere in Multi-App-Szenarien wichtig, um Seiteneffekte zu vermeiden.

7.6 PARAMETRISIERUNG UND GRENZFÄLLE

Mike Wild

Blockgröße und Format Die Wahl $N = 1024$ Frames bei 48 kHz reduziert Aufruf-Overhead und hält die Latenz moderat. Float32 vermeidet Quantisierungsartefakte vor dem Downsampling im `WavWriterThread` (vgl. Kapitel 9) und sorgt dafür, dass das Audio in hoher Qualität gespeichert werden kann.

Pegelsteuerung `sysGain` und `micGain` werden zur Laufzeit aus `QSettings` gelesen und erlauben eine grobe Balance. Das `qBound`-Clipping schützt vor Übersteuerung in der Mischstufe.

Fehlerszenarien Typische Fehler sind (I) fehlende Standard-Geräte in `pactl info`, (II) fehlende Rechte zum Laden von Modulen, (III) konkurrierende PulseAudio-Clients. Alle Fälle führen zu `return false` in `initializeCapture()` nach einem gezielten `cleanupCapture()`.

8

AUDIOAUFNAHME UNTER MACOS

Pakize Gökkaya

Die digitale Audioverarbeitung hat in den letzten Jahren eine immer größere Bedeutung erlangt, sowohl im professionellen als auch im privaten Umfeld. Anwendungen reichen von Musikproduktion, Podcast-Aufnahme, Videokonferenzen bis hin zu Sprachsteuerungssystemen und Machine-Learning-gestützter Audioanalyse. Für die plattformübergreifende Softwareentwicklung stellt insbesondere die Audioaufnahme eine technische Herausforderung dar, da sich die zugrunde liegenden Audio-APIs und Treiberarchitekturen zwischen Betriebssystemen wie Windows, macOS und Linux erheblich unterscheiden.

Im Rahmen dieser Arbeit wurde die Klasse MacCaptureThread entwickelt, um unter macOS eine stabile, latenzarme und verlustfreie Audioaufnahme zu ermöglichen. Diese Implementierung basiert auf Qt – einem plattformübergreifenden Framework – und nutzt die Klasse QAudioSource zur Erfassung von Audiodaten in Echtzeit. Besonderes Augenmerk wird auf die Integration des virtuellen Audiotreibers BlackHole gelegt, der als Schnittstelle dient, um Systemaudio direkt abzugreifen, ohne physische Mikrofone oder externe Geräte zu verwenden.

8.1 EINFÜHRUNG IN DIE AUDIOARCHITEKTUR VON MACOS

Pakize Gökkaya

Die Audioverarbeitung unter macOS unterscheidet sich grundlegend von derjenigen unter Windows oder Linux. Apple setzt seit vielen Jahren auf das sogenannte Core Audio Framework, das eine hochperformante, hardwarenahe Schnittstelle für die Ein- und Ausgabe von Audiosignalen bereitstellt. Core Audio ist vollständig in das Betriebssystem integriert und bietet niedrige Latenzen sowie eine präzise Steuerung von Audioströmen. Im Gegensatz zu Windows, wo die Audioarchitektur durch mehrere parallele APIs wie WASAPI, DirectSound oder ASIO geprägt ist, und Linux, das in der Praxis meist auf PulseAudio oder ALSA setzt, verfolgt macOS einen stark zentralisierten Ansatz.

Eine Besonderheit von macOS besteht darin, dass Systemaudio, also die Gesamtausgabe aller Anwendungen, aus Datenschutz- und Sicherheitsgründen nicht direkt abgegriffen werden kann. Apple verhindert so, dass Programme ohne Wissen der Nutzer Audioinhalte mitschneiden. Aus diesem Grund ist für die Aufnahme von Systemaudio zwingend die Verwendung sogenannter virtueller Audiotreiber notwendig. Diese Treiber stellen dem Betriebssystem ein „virtuelles Audiogerät“ bereit, über das Audio von einer Anwendung zur anderen geleitet werden kann. Genau hier kommt in der Praxis das Open-Source-Werkzeug BlackHole zum Einsatz.

8.2 technischer hintergrund

Pakize Gökkaya

1. Audio-APIs unter macOS

   macOS bietet mit Core Audio eine leistungsfähige, aber komplexe Low-Level-API, die direkten Zugriff auf die Audiohardware ermöglicht. Die direkte Programmierung gegen Core Audio ist jedoch für plattformübergreifende Anwendungen aufwendig. Aus diesem Grund verwendet MacCaptureThread die Qt-eigene Abstraktion QAudioSource, die intern Core Audio anspricht, aber eine einheitliche Schnittstelle für Windows, macOS und Linux bereitstellt. [2]

2. QAudioSource und QIODevice

   Die Klasse QAudioSource ist darauf ausgelegt, kontinuierlich Audiodaten von einer Eingangsquelle zu erfassen. Die Ausgabe erfolgt in ein QIODevice, das entweder synchron (durch blockierendes Lesen) oder asynchron (mittels readyRead-Signal) ausgelesen werden kann. Der Ansatz in MacCaptureThread verwendet asynchrones Lesen, um eine konstante, latenzarme Verarbeitung zu gewährleisten und UI-Blockaden zu vermeiden. [6]

3. Signal-Slot-Mechanismus in Qt

   Das Zusammenspiel zwischen QAudioSource und der Anwendung erfolgt über den Signal-Slot-Mechanismus von Qt. Immer wenn neue Audiodaten verfügbar sind, löst das QIODevice ein readyRead-Signal aus. Dieses wird mit einer Lambda-Funktion verbunden, die die Daten sofort ausliest, in einem internen Puffer (audioBuffer) speichert und über das Signal audioDataReady an andere Teile der Anwendung weitergibt. [6]

8.3 implementierung von maccapturethread

Pakize Gökkaya

Die Implementierung der Klasse `MacCaptureThread` stellt eine spezialisierte Lösung zur Audioaufnahme unter macOS dar. Sie basiert auf der von uns definierten generischen Basisklasse `CaptureThread`, die eine plattformunabhängige Schnittstelle zur Verfügung stellt und von der verschiedene Betriebssystem-spezifische Ableitungen erstellt werden können. Die macOS-Variante unterscheidet sich dabei in mehrfacher Hinsicht von Windows- oder Linux-Implementierungen, da sie auf die Eigenheiten des Apple-Frameworks *Core Audio* sowie auf die Qt-Multimedia-Schnittstelle (`QAudioSource`) zurückgreift.

Im Folgenden werden die wesentlichen Schritte der Implementierung erläutert und durch zusätzliche technische Hintergründe ergänzt.

8.3.1 *Initialisierung der Audioaufnahme*

Die Methode `initializeCapture()` übernimmt die Konfiguration und Inbetriebnahme der Audioquelle. Dabei werden mehrere Schritte durchlaufen:

1. **Instanziierung von QAudioSource**: Zunächst wird ein Objekt vom Typ `QAudioSource` erzeugt, dem ein zuvor spezifiziertes Audioformat übergeben wird. Zu den typischen Parametern gehören die Abtastrate (z. B. 44,1 kHz oder 48 kHz), die Anzahl der Kanäle (Mono, Stereo oder Mehrkanal) sowie das Sample-Format (z. B. 16-bit Integer). Diese Parameter sind für die Kompatibilität mit nachgelagerten Verarbeitungsschritten entscheidend.

2. **Öffnen des Eingabegeräts**: Anschließend wird ein Eingabegerät (`QIODevice`) über die `QAudioSource` geöffnet. Dieses Eingabegerät repräsentiert den kontinuierlichen Datenstrom der Audiodaten, der von Hardware oder virtuellen Treibern wie „BlackHole“ bereitgestellt wird.

3. **Signal-Slot-Mechanismus mit Lambda-Funktion**: Das Signal `readyRead()` des Eingabegeräts wird mit einer Lambda-Funktion verknüpft. Diese Lambda-Funktion liest die aktuell verfügbaren Audiodaten aus und speichert sie in einem internen Puffer. Dieser Ansatz vermeidet Polling-Mechanismen und sorgt für eine asynchrone, ressourcenschonende Verarbeitung. Die Verwendung von Lambdas bietet zudem eine enge Kapselung und vermeidet die Notwendigkeit zusätzlicher Hilfsmethoden.

Die Initialisierung stellt damit sicher, dass unmittelbar nach dem Start des Threads ein funktionierender und kontinuierlicher Datenfluss gewährleistet ist.

8.3.2 *Kontinuierliche Aufnahme*

Die Methode `captureLoopIteration()` übernimmt die eigentliche Verarbeitungsschleife. Im Unterschied zu klassischen Implementierungen, die auf einer aktiven Schleife basieren, delegiert `QAudioSource` die Steuerung des Datenflusses an das Betriebssystem. Neue Iterationen erfolgen ausschließlich dann, wenn tatsächlich Audiodaten im Eingabepuffer vorliegen. Dadurch sinkt die CPU-Last erheblich, was insbesondere bei Echtzeitanwendungen relevant ist. Zusätzlich trägt dieses Verfahren zu einem deterministischeren Timing bei, da die Aufnahme nicht von manuell gesetzten Sleep- oder Polling-Intervallen abhängt, sondern direkt durch Hardware-Events getriggert wird.

Ein weiterer Vorteil dieser Architektur besteht in der einfachen Integration in Qt's Event-Loop, wodurch die Aufnahme nahtlos mit anderen GUI- oder Netzwerk-Operationen koexistieren kann.

8.3.3 *Pufferverwaltung und Datenzugriff*

Die aufgezeichneten Daten werden in einem internen `QByteArray`-Puffer (`audioBuffer`) zwischengespeichert. Über die Methode `getBuffer()` kann auf diesen Puffer zugegriffen werden. Dies erlaubt eine klare Trennung zwischen der Erfassungsschicht (Capture) und der Verarbeitungsschicht (z. B. Speicherung, Streaming oder Analyse). Ein typisches Anwendungsbeispiel wäre die direkte Übergabe des Puffers an eine Fourier-Transformation zur Frequenzanalyse, an einen Netzwerk-Stack für Live-Streaming oder an eine Datei-Engine zur persistenten Speicherung im WAV- oder MP3-Format.

Das Signal `audioDataReady()` informiert verbundene Komponenten unmittelbar über neue Daten. Dies entspricht dem „Observer Pattern“ und erleichtert die lose Kopplung verschiedener Systemkomponenten.

8.3.4 *Ressourcenmanagement und Bereinigung*

Die Methode `cleanupCapture()` ist verantwortlich für die Freigabe sämtlicher Ressourcen. Hierbei wird sichergestellt, dass sowohl die `QAudioSource` als auch das zugehörige Eingabegerät korrekt geschlossen werden. Dies verhindert Speicherlecks und blockierte Hardwaregeräte. Besondere Bedeutung hat dies im Fehlerfall oder beim kontrollierten Abbruch einer Aufnahme. Ein sauberer Shutdown ist Voraussetzung dafür, dass die Audiohardware unmittelbar für weitere Prozesse zur Verfügung steht.

8.3.5 *Besonderheiten unter macOS*

Die Implementierung unter macOS unterscheidet sich von anderen Plattformen durch die Abhängigkeit von *Core Audio*, das von Qt abstrahiert wird. Während unter Windows oftmals *WASAPI* oder *DirectSound* verwendet werden, setzt Linux auf Treiberarchitekturen wie ALSA oder PulseAudio. Unter macOS kommt hinzu, dass virtuelle Audiogeräte wie *BlackHole* oder *Soundflower* benötigt werden, um Systemaudio überhaupt aufzuzeichnen. Diese Treiber simulieren ein Eingabegerät, das den Systemton als Datenstrom bereitstellt. Besonders relevant ist hierbei die Unterscheidung zwischen „BlackHole 2ch“ und „BlackHole 16ch“: Erstere bietet lediglich zwei Kanäle (Stereo), während letztere bis zu 16 Kanäle parallel abbilden kann. Die Wahl des Treibers beeinflusst daher maßgeblich die Flexibilität bei Mehrkanalanwendungen, etwa für Audio-Engineering oder Live-Mischungen.

8.4 VERWENDUNG VON BLACKHOLE ALS AUDIOQUELLE

Pakize Gökkaya

Ein zentrales Element der Implementierung ist die Verwendung von BlackHole, einem virtuellen Audio-Treiber für macOS. BlackHole ermöglicht es, den Systemton (alles, was über die Lautsprecher ausgegeben würde) als Eingangssignal abzugreifen. Die Gründe für den Einsatz von BlackHole statt anderer Lösungen wie Soundflower oder Loopback sind:

- Aktive Weiterentwicklung und Kompatibilität mit aktuellen macOS-Versionen (inkl. Apple Silicon).
- Geringe Latenz durch native Core-Audio-Integration.
- Unterstützung für Mehrkanalbetrieb (2-Kanal- und 16-Kanal-Versionen verfügbar).
- Kostenlos und Open Source. [1]

Unterschiede zwischen 2-Kanal- und 16-Kanal-Variante

- 2-Kanal-Version: Für Standard-Stereoaufnahmen geeignet; geringerer Ressourcenverbrauch; einfachere Weiterverarbeitung.
- 16-Kanal-Version: Erlaubt paralleles Capturing von mehreren unabhängigen Audioströmen; ideal für komplexe Setups in Musikproduktion oder Live-Mischungen.

In MacCaptureThread kann je nach Anwendungsfall das gewünschte virtuelle Gerät als Eingangsquelle ausgewählt werden.

8.5 VERGLEICH ZU WINDOWS-IMPLEMENTIERUNGEN

Pakize Gökkaya

Unter Windows wird Audio-Capturing in Qt häufig über QAudioInput realisiert, das wiederum auf WASAPI (Windows Audio Session API) oder MME (MultiMedia Extensions) aufsetzt. Die wesentlichen Unterschiede zu macOS sind:

- Gerätemanagement: Windows erlaubt explizite Auswahl des „Loopback"-Modus für WASAPI, um Systemaudio direkt zu erfassen. Unter macOS ist dies ohne virtuelle Treiber wie BlackHole nicht möglich.
- Latenzverhalten: Core Audio unter macOS bietet in der Regel geringere Latenzen als WASAPI, wenn beide optimal konfiguriert sind.
- Treiberabhängigkeit: Während Windows-Systeme oft mit Onboard-Audio und Loopback-Support arbeiten, muss macOS ohne BlackHole auf externe Hardware zurückgreifen, um Systemaudio zu erfassen. [3]

8.6 HERAUSFORDERUNGEN UND LÖSUNGSANSÄTZE

Pakize Gökkaya

Die Implementierung von MacCaptureThread war mit mehreren Herausforderungen verbunden:

1. Geräteauswahl in QAudioSource
    - Herausforderung: BlackHole muss gezielt aus der Liste verfügbarer Eingabegeräte ausgewählt werden.
    - Lösung: Abfrage aller verfügbaren Eingabegeräte und explizite Auswahl des BlackHole-Devices per QMediaDevices.
2. Thread-Sicherheit
    - Herausforderung: Gleichzeitiger Zugriff auf den audioBuffer kann zu Race Conditions führen.
    - Lösung: Puffern der Daten in QByteArray mit Mutex-Schutz, um Race Conditions beim Lesen und Schreiben zu vermeiden.
3. Ressourcenfreigabe
    - Herausforderung: Abbrüche oder Exceptions können zu nicht freigegebenen Ressourcen führen.
    - Lösung: Sicherstellen, dass cleanupCapture() auch bei Exceptions oder erzwungenem Thread-Abbruch ausgeführt wird.

8.6.1 *Fazit*

Pakize Gökkaya

Die Klasse `MacCaptureThread` stellt eine robuste und erweiterbare Lösung dar, um Audioaufnahmen unter macOS zu realisieren. Durch die Kombination von `QAudioSource`, `QIODevice` und Qt's Signal-Slot-Mechanismus wird eine asynchrone, latenzarme und ressourcenschonende Aufnahme ermöglicht. Im Vergleich zu Windows oder Linux erfordert macOS zusätzliche virtuelle Treiber, was die Implementierung komplexer macht. Gleichzeitig bietet der Ansatz aber auch eine hohe Flexibilität und eine klare Trennung zwischen Erfassung, Pufferung und Weiterverarbeitung. Damit eignet sich `MacCaptureThread` sowohl für einfache Stereoaufnahmen als auch für komplexe Mehrkanalszenarien.

9

WAV-SCHREIBTHREAD (PUFFERUNG, DOWNSAMPLING UND DATEIFORMAT)

Mike Wild

Der `WavWriterThread` entkoppelt die zeitkritische Audioaufnahme von der persistenten Speicherung. Er implementiert ein *Producer–Consumer*-Muster: Capture-Threads liefern asynchron `QList<float>`-Blöcke (Stereo, 48 kHz, Float32), der Writer-Thread puffert diese, schreibt periodisch in zwei WAV-Dateien und signalisiert den Abschluss der Session.

9.1 Ziele und Randbedingungen

Mike Wild

- **Zwei parallele Ausgaben:** (I) *HQ-Datei* in 48 kHz, Stereo, 32-bit Float (für Archivierung/Weiterverarbeitung) und (II) *ASR-Datei* in 16 kHz, Mono, 16-Bit-Integer (für Spracherkennung) speichern.
- **Latenz/IO-Last balancieren:** Das Schreiben erfolgt blockweise, gesteuert über einen konfigurierbaren *Flush-Threshold* in Byte (Default via `QSettings`).
- **Robuste Finalisierung:** WAV-Header werden erst am Ende korrekt gefüllt, nachdem die exakte Datenlänge bekannt ist; während der Aufnahme stehen Platzhalter im Datei-Header.

9.2 Lebenszyklus und Synchronisation

Mike Wild

Die externe API umfasst:

- `startWriting(hqPath, asrPath)`: Dateien öffnen, Headerplatzhalter schreiben, Thread aktivieren.
- `writeChunk(QList<float>)`: Producer-Slot; fügt Audio-Daten-Blöcke threadsicher in den Puffer ein.
- `stopWriting()`: beendet die Session; restliche Daten werden geschrieben, Header finalisiert.
- `shutdown()`: beendet den Thread (inkl. `wait()`) sicher.

9.3 Downsampling und Formatkonvertierung

Mike Wild

Die HQ-Datei erhält den Float32-Stereostream unverändert. Für die ASR-Datei wird der Stream *downmixed* und *downsampled*:

1. **Stereo** $\rightarrow$ **Mono:** Mittelwertbildung pro Frame $m = \frac{1}{2}(L + R)$.

2. **48 kHz → 16 kHz:** Faktorielles Downsampling mit Faktor 3: pro drei Eingangssamples wird genau ein Ausgangssample erzeugt. Ein *Offset*-Akkumulator gewährleistet, dass Chunk-Grenzen korrekt fortgeführt werden:

$$\texttt{m\_downsampleOffset} \leftarrow (N_{\text{frames}} + \texttt{m\_downsampleOffset}) \bmod 3.$$

3. **Float32 → Int16:** Skalierung $[-1, 1]_{\text{Float}_{32}} \mapsto [-32768, 32767]_{\text{16-Bit-Integer}}$

9.4 WAV-HEADER-STRATEGIE

Mike Wild

Beim Start werden 44 Null-Bytes als Header-Platzhalter geschrieben. Nach Abschluss der Session werden die korrekten Header an Datei-Offset 0 (Dateianfang) nachgetragen:

- HQ: `AudioFormat=3` (IEEE Float), Stereo, 48 kHz, 32 bit.
- ASR: `AudioFormat=1` (PCM), Mono, 16 kHz, 16 bit.

Die Felder `RIFF–ChunkSize` und `data–Subchunk2Size` ergeben sich aus den gezählten Nutzdatenbytes (`m_hqBytesWritten`, `m_asrBytesWritten`).

9.5 KONFIGURIERBARKEIT UND ROBUSTHEIT

Mike Wild

Flush-Threshold. Die Mindestpuffergröße, die erreicht werden muss, bevor ein Dateischreibvorgang erfolgt, ist über `QSettings` konfigurierbar (im *SettingsWizard*); so lässt sich das Verhältnis von Latenz zu Schreib-Zugriff-Last steuern.

Atomare Zustände. Flags `m_active` und `m_shutdown` werden atomar verwaltet; `QWaitCondition`–Signale trennen *Start-/Stop*-von *Datenwarte*-Phasen. Die Finalisierung schreibt verbleibende Pufferdaten sicher und füllt erst dann die Header.

9.6 ZUM FORMAT

Mike Wild

Die Entscheidung für Float32 als *Transportformat* bis zum Writer minimiert Rundungsfehler in der Mischphase und vermeidet Clipping-Artefakte vor der finalen Quantisierung. Die einfache 3:1-Reduktion ist für ASR robust und rechenarm; für höhere Qualität könnte optional ein Polyphasen- oder Windowed-Sinc-Resampler integriert werden. Die getrennten Dateien erfüllen zwei Zwecke: *Nutzung* für die ASR (ASR-Datei) und *Archivierung/Analyse* des Meetings und nachträgliche Korrektur des Transkripts (HQ-Datei).

10 PYTHON-INTEGRATION (ENVIRONMENT & ASR-PROZESS)

Mike Wild

Die Anwendung integriert einen externen Python-Workflow für die automatische Spracherkennung (ASR) und einen für die Tag-Erstellung mittels `spaCy`. Drei Komponenten kapseln diese Integration:

- **PythonEnvironmentManager:** Überprüfung/Installation einer isolierten Python-Umgebung (virtuelle Umgebung), inkl. (Neu-)Installation benötigter Pakete und Nutzerführung via Dialog.
- **AsrProcessManager:** Startet und überwacht den Python-Prozess (QProcess), liest fortlaufend die Ergebnis-Ausgaben und speist diese als `MetaText`-Segmente in das Datenmodell `Transcription` ein.
- **TagGeneratorManager:** Startet und überwacht den Python-Prozess (QProcess), übergibt den Inhalt des Transkripts, nimmt die Tags entgegen und informiert den Nutzer über eine `QMessageBox`.

10.1 ZIELE UND ABGRENZUNG

Mike Wild

- **Reproduzierbarkeit:** konsistente Python-Version und Paketstände unabhängig vom System.
- **Robuste Prozesssteuerung:** klares Start/Stop, Fehlerpfade (Exitcodes, Fehlermeldungen), thread-sichere Weitergabe der Ergebnisse an die GUI.
- **Entkopplung:** C++/Qt und Python sind sauber getrennt; Kommunikation nur über Prozessgrenzen.

10.2 PYTHONENVIRONMENTMANAGER

Mike Wild

Der `PythonEnvironmentManager` kapselt die Lebenszyklus-Operationen der Python-Umgebung: Existenzprüfung, Neuinstallation, Paketinstallation und die Nutzerinteraktion über einen Installationsdialog. Pfade und Zustände werden in den globalen Einstellungen verwaltet (`QSettings`), sodass einerseits andere Komponenten (z. B. `AsrProcessManager`) die Interpreter-/Skriptpfade aus den Settings laden können und andererseits die Skripte zur Laufzeit vom Nutzer ausgetauscht werden können.

Aufgabenüberblick

- Prüfen, ob eine virtuelle Umgebung vorhanden und funktionsfähig ist (Interpreter aufrufbar, Pip/Pakete verfügbar).
- Optionales *Reinstall*: bestehende Umgebung rekursiv löschen und neu anlegen.
- Installation der benötigten Pakete (z. B. via `pip install -r requirements.txt`).
- Rückmeldung des Ergebnisses (Erfolg/Fehlertext) an den aufrufenden Kontext.

10.3 ASRPROCESSMANAGER

Mike Wild

Der `AsrProcessManager` steuert den externen Python-Prozess zur Transkription (ASR). Er übernimmt die Pfadauflösung (`loadPaths()`), den Start der Transkription für eine gegebene WAV-Datei und das Parsen der Ergebnisausgaben.

Signale und zentrale Methoden

Mike Wild

`startTranscription(wavPath)` Startet den QProcess (Python + ASR-Skript) für die angegebene WAV-Datei.

`stop()` Beendet den laufenden Prozess (sanft/sofort, je nach Zustand).

`parseLine(line)` Parst eine Textzeile aus der Prozessausgabe in einen `MetaText`.

`loadPaths()` Lädt Interpreter-/Skriptpfade aus `QSettings`.

`segmentReady(meta)` Signal – Wird für jedes erkannte Segment emittiert; `meta` ist ein `MetaText`.

`finished(success, err)` Signal – Markiert Prozessende (Exitcode=0 ⇒ `success=true`); bei Fehlern transportiert `err` die Ursache.

Prozessfluss und Interprozesskommunikation

Mike Wild

Das ASR-Python-Skript wird als externer Prozess ausgeführt (`QProcess`). Die Ergebnisse werden *zeilenweise* über die Standardausgabe geliefert; jede Zeile repräsentiert ein erkennbares Segment und wird mittels `parseLine()` in das interne `MetaText`-Format überführt. Während der Laufzeit emittiert der Manager `segmentReady(meta)`, sodass das Modell `Transcription` inkrementell

wachsen kann. Bei Prozessende wird `finished(success, errorMsg)` gesendet.

10.4 FEHLERBEHANDLUNG

Mike Wild

Exitcodes und Fehlermeldungen Der `AsrProcessManager` unterscheidet zwischen regulärem Abschluss (Exitcode 0) und Fehlerfällen. Bei Fehlern wird der in Stderr gesammelte Text als `errorMsg` via `finished(false, errorMsg)` weitergereicht.

Abbruch `stop()` beendet einen laufenden Prozess kontrolliert. Auf GUI-Seite ist `onStartClicked()` defensiv: ein ggf. vorher laufender ASR-Prozess wird zu Beginn gestoppt, um Kollisionen zu vermeiden.

Parsing-Fehler `parseLine()` kapselt die Übersetzung einer Prozesszeile in `MetaText`. Fehlerhafte Zeilen werden verworfen oder als diagnostischer Hinweis protokolliert, ohne den Gesamtprozess zu unterbrechen.

10.5 TAGGENERATORMANAGER

Mike Wild

Der `TagGeneratorManager` kapselt die Python-basierte Schlagwortanalyse (z. B. mittels `spaCy`). Er startet einen externen Prozess, übergibt den Transkripttext und erhält eine Liste von Tags zurück. Die Ergebnisse werden als Signal an die GUI/Modellschicht weitergereicht (vgl. `MainWindow::onGenerateTags()` und die Signalverknüpfung in `doConnects()`).

Signale und zentrale Methoden

`generateTagsFor(text)` Startet den QProcess (Python + Tagging-Skript) und übergibt den vollständigen Transkripttext.

`stop()` Beendet einen laufenden Tagging-Prozess kontrolliert (z. B. vor einem neuen Run).

`loadPaths()` Lädt Interpreter-/Skriptpfade aus `QSettings` (z. B. `pythonPath`, `tagScriptPath`).

`parseOutput(data)` Übersetzt die Prozessausgabe (z. B. JSON- oder zeilenweise Liste) in eine `QStringList`.

`tagsReady(tags, success, err)` Signal — Liefert die extrahierten Tags sowie Erfolg/Fehlertext zurück.

Prozessfluss und Interprozesskommunikation

Mike Wild

Die Tag-Generierung folgt einem einfachen Request–Response-Muster: Die C++-Seite übergibt den kompletten Transkripttext, die Python-Seite liefert eine normalisierte Liste von Schlagwörtern zurück. Typische Normalisierungsschritte (auf der Python-Seite) umfassen Kleinschreibung, Trimmen, Duplikatentfernung und ggf. Lemmatisierung; das konkrete Verfahren ist vom verwendeten Skript abhängig.

Integration in die GUI und das Datenmodell

Mike Wild

Die GUI löst die Analyse über den entsprechenden Button aus:

- **Aufruf**: `onGenerateTags()` prüft, ob Text vorhanden ist, deaktiviert den Button, zeigt Status an und ruft `generateTagsFor(m_script->text())` auf.
- **Ergebnis**: Auf `tagsReady(tags, success, err)` setzt die GUI bei Erfolg die Tags in `Transcription` (`m_script->setTags(tags)`) und zeigt die Liste an; bei Fehlern wird eine Warnung angezeigt. Der Button wird wieder aktiviert.

Fehler- und Qualitätsaspekte

Mike Wild

- **Timeout/Abbruch:** Lange Analysen (sehr große Transkripte) sollten durch einen Prozess-Timeout bzw. `stop()` abfangbar sein.
- **Speicherbedarf:** Für sehr große Texte empfiehlt sich eine Übergabe per temporärer Datei statt `stdin`, um Kopier-Overhead zu reduzieren.
- **Normalisierung:** Deduplizieren, Kleinschreibung und Längenfilter (z. B. ≥ 2 Zeichen) erhöhen die Qualität der Tag-Liste; diese Schritte sollten im Python-Skript konsistent implementiert sein.
- **Determinismus:** Für reproduzierbare Ergebnisse sollten nicht-deterministische Komponenten (z. B. zufällige Scores) vermieden oder mit fixem Seed betrieben werden.

11 TRANSKRIPTION UND SPRECHERDIARISIERUNG

Fabian Scherer

Die automatische Spracherkennung (ASR) und die Sprecherdiarisierung (SD) sind zentrale Komponenten dieses Projekts, die es ermöglichen, kontinuierliche Audiodaten in eine textbasierte und sprecherbasierte Ausgabe umzuwandeln. Beide Funktionen laufen in einem separaten Python-Backend, das in Echtzeit mit dem C++-Frontend über Standard-Pipes kommuniziert.

11.1 AUDIODATEN-PIPELINE

Fabian Scherer

Die Audiodaten werden im C++-Frontend erfasst und als normalisierte 32-Bit-Float-Werte im Bereich von -1.0 bis 1.0 verarbeitet. Diese werden kontinuierlich in binärer Form an das Python-Backend gesendet.

Auf der Python-Seite werden die empfangenen Byte-Daten chunkweise ausgelesen, wobei ein Chunk einer Sekunde entspricht, in NumPy-Arrays umgewandelt und an die jeweiligen Verarbeitungsmodule (ASR und SD) weitergeleitet.

11.2 AUTOMATISCHE SPRACHERKENNUNG (ASR)

Fabian Scherer

Für die ASR-Funktionalität wird ein vortrainiertes **Wav2Vec2-Modell** verwendet, das für die deutsche Sprache optimiert ist. Das Modell ist auf eine Samplerate von 16 kHz ausgelegt, weshalb die eingehenden Audiodaten auf diese Rate resampelt werden, um eine hohe Genauigkeit zu gewährleisten.

Der Verarbeitungsprozess gliedert sich wie folgt:

- **1. Input-Verarbeitung:** Jeder 1-Sekunden-Audio-Chunk wird direkt in einen Tensoren umgewandelt.
- **2. Inferenz:** Der Tensor wird durch das Wav2Vec2-Modell geleitet, welches eine Sequenz von Logits generiert.
- **3. Dekodierung:** Ein **Language Model (KenLM)** wird verwendet, um die Logits in die wahrscheinlichste Wortsequenz zu dekodieren. Dieses Modell verbessert die Transkriptionsqualität erheblich, indem es den Kontext und die grammatikalische Struktur der Sprache berücksichtigt.
- **4. Ausgabe:** Der transkribierte Text wird als JSON-Nachricht zurück an das C++-Frontend gesendet.

11.2.1 *Das verwendete Modell*

Die automatische Spracherkennung (ASR) in diesem Projekt basiert auf dem Wav2Vec2-Modell, insbesondere der für das Deutsche angepassten Variante aware-ai/wav2vec2-large-xlsr-53-german-with-lm (aware-ai, n.d.). Wav2Vec2 stellt einen signifikanten Fortschritt in der ASR-Forschung dar, indem es die Notwendigkeit großer, manuell transkribierter Korpora für das Training reduziert. Die Architektur zeichnet sich durch einen zweistufigen Lernprozess aus: selbstüberwachtes Vor-Training auf unbeschrifteten Audiodaten und überwachtes Fein-Tuning für eine spezifische Transkriptionsaufgabe (Baevski et al., 2020).

11.2.1.1 *Vor-Training (Pre-training)*

Die Grundlage des verwendeten Modells bildet die Wav2Vec2-XLS-R-53-Architektur. Im Vor-Trainingsschritt lernt das Modell die intrinsische Struktur und linguistische Eigenheiten von Sprache durch ein selbstüberwachtes Lernverfahren. Hierbei werden große Mengen unbeschrifteter Audiodaten als Input verwendet. Der Prozess kann wie folgt beschrieben werden: Das Modell maskiert (verdeckt) Teile des Audio-Eingangs und lernt anschließend, diese maskierten Segmente aus dem umliegenden Kontext vorherzusagen. Dieser Ansatz ermöglicht es dem Modell, robuste und kontextreiche Repräsentationen von Sprachlauten zu extrahieren, ohne semantisches oder linguistisches Wissen zu benötigen. Die Basisvariante wurde auf einem umfangreichen Datensatz von 436.000 Stunden unbeschrifteter Audiodaten in 53 verschiedenen Sprachen vor-trainiert (Baevski et al., 2020).

11.2.1.2 *Feinabstimmung (Fine-tuning)*

Nach dem Vor-Training wurde das Modell aware-ai/wav2vec2-large-xlsr-53-german-with-lm für die automatische Spracherkennung der deutschen Sprache feinabgestimmt. In dieser Phase wird der vortrainierte Feature-Extraktor mit einem spezifischen Klassifikator verbunden, um Audiodaten direkt in Text zu transkribieren. Ein entscheidendes Merkmal dieses Modells ist die Integration eines KenLM-Sprachmodells (Language Model). Dieses LM wird zur Dekodierung der Modellausgaben verwendet, um die Wahrscheinlichkeit von Wortsequenzen zu bewerten und so die grammatikalisch und semantisch plausibelste Transkription zu ermitteln. Die Kombination eines leistungsstarken, vor-trainierten Modells mit einem sprachspezifischen LM ermöglicht eine hohe Transkriptionsgenauigkeit und Robustheit gegenüber akustischen und linguistischen Variationen.

11.3 SPRECHERDIARISIERUNG (SD)

Fabian Scherer

Die Sprecherdiarisierung hat die Aufgabe, zu erkennen, wann welcher Sprecher spricht. Dies ist besonders nützlich bei Gesprächen mit mehreren Teilnehmern.

Im Gegensatz zur ASR, die in Echtzeit auf jedem 1-Sekunden-Chunk arbeitet, verarbeitet die Diarisierung Audio in längeren Segmenten. Ein separater Puffer sammelt drei aufeinanderfolgende 1-Sekunden-Chunks, um eine 3-Sekunden-Analyse durchzuführen.

Der Prozess umfasst:

- **1. Chunk-Akkumulation:** Drei 1-Sekunden-Audio-Chunks werden zu einem 3-Sekunden-Segment zusammengefügt.
- **2. Sprechereinbettung:** Das Audiosignal wird durch ein vortrainiertes Modell (wie z.B. Pyannote.audio) geleitet, um eine kompakte Darstellung der Stimme des Sprechers (eine sogenannte SSprechereinbettung") zu erzeugen.
- **3. Clustering:** Die Sprechereinbettung wird analysiert und mit zuvor erkannten Sprechern verglichen. Das Modell weist dem Segment einen bestimmten Sprecher zu oder identifiziert einen neuen.
- **4. Ausgabe:** Das Ergebnis der Diarisierung (z.B. SSprecher A hat von Sekunde 0 bis 3 gesprochen") wird als JSON-Nachricht an das C++-Frontend übermittelt.

12 DATENBANK

Yolanda Hadiana Fiska

Die Integration einer Datenbank stellt einen zentralen Bestandteil der Gesamtlösung dar. Sie bildet die Grundlage für eine effiziente Speicherung, Verwaltung und spätere Analyse der im Projekt erzeugten Transkriptionsdaten.

12.1 EINLEITUNG

Yolanda Hadiana Fiska

Die Nutzung einer durchsuchbaren Datenbank ist notwendig, da Meetings und Konferenzen oft eine hohe Informationsdichte aufweisen. Ohne eine strukturierte Ablage wäre es für Teilnehmer nahezu unmöglich, spezifische Aussagen oder Diskussionen effizient wiederzufinden. Eine durchsuchbare Datenbank ermöglicht daher nicht nur die automatisierte Dokumentation, sondern auch eine signifikante Verbesserung der Nachverfolgbarkeit und Wissenssicherung.

Für die Datenbankintegration ergeben sich folgende zentrale Anforderungen:

- **Speicherung:** Persistente Ablage der Transkripte in einer konsistenten Datenstruktur, die Metadaten zu Meetings und Sprechern einschließt.
- **Durchsuchbarkeit:** Unterstützung effizienter Volltextsuche mit sprachspezifischer Tokenisierung, Normalisierung und optionaler Fuzzy-Suche.
- **Skalierbarkeit:** Möglichkeit, auch große Datenmengen aus zahlreichen Meetings performant zu verwalten.
- **Echtzeit-Zugriff:** Sofortige Verfügbarkeit neu erfasster Transkriptionssegmente für nachfolgende Suche und Analyse.

Das primäre Ziel der Datenbankintegration besteht in der **Ermöglichung einer strukturierten Ablage und einer effizienten Suche** innerhalb von Transkripten. Damit wird die Grundlage geschaffen, um Inhalte nicht nur zu dokumentieren, sondern sie auch als nachhaltige Wissensbasis für Teilnehmer und Organisationen nutzbar zu machen.

12.2 DATENMODELLIERUNG

Yolanda Hadiana Fiska

Die Datenbankstruktur ist so gestaltet, dass sie sowohl die Speicherung der Transkripte als auch deren effiziente Durchsuchbarkeit und Kontextualisierung ermöglicht. Im Zentrum stehen drei Haupttabellen, deren Beziehungen im folgenden erläutert werden.

12.2.1 *Wichtige Tabellen*

Yolanda Hadiana Fiska

- **Besprechungen (`besprechungen`):** Enthält Metadaten zu einzelnen Meetings. Dazu gehören eine eindeutige ID, der Titel der Besprechung sowie das Erstellungsdatum. Diese Tabelle dient als Ankerpunkt für alle weiteren Informationen.
- **Sprecher (`sprecher`):** Speichert die Namen der an Meetings beteiligten Sprecher. Über das Fremdschlüsselfeld `besprechungen_id` wird eine direkte Beziehung zur jeweiligen Besprechung hergestellt.
- **Aussagen (`aussagen`):** Enthält die eigentlichen Transkriptionsabschnitte. Neben den Roh- und verarbeiteten Texten werden hier Start- und Endzeitstempel, optionale Schlagwörter (`tags`), sowie ein generierter `tsvector` für die Volltextsuche abgelegt. Über Fremdschlüssel zu `sprecher` und `besprechungen` sind die Aussagen eindeutig einem Sprecher und einer Besprechung zugeordnet.

12.2.2 *Beziehungen zwischen den Tabellen*

Yolanda Hadiana Fiska

Zwischen den Tabellen bestehen folgende Beziehungen:

- **1:n Beziehung zwischen `besprechungen` und `sprecher`:** Eine Besprechung kann mehrere Sprecher enthalten, ein Sprecher ist jedoch genau einer Besprechung zugeordnet.
- **1:n Beziehung zwischen `besprechungen` und `aussagen`:** Jede Besprechung kann mehrere Aussagen beinhalten.
- **1:n Beziehung zwischen `sprecher` und `aussagen`:** Ein Sprecher kann im Verlauf einer Besprechung mehrere Aussagen tätigen.

12.2.3 *Visualisierung*

Yolanda Hadiana Fiska

Das folgende ER-Diagramm veranschaulicht die Relationen zwischen den Tabellen:

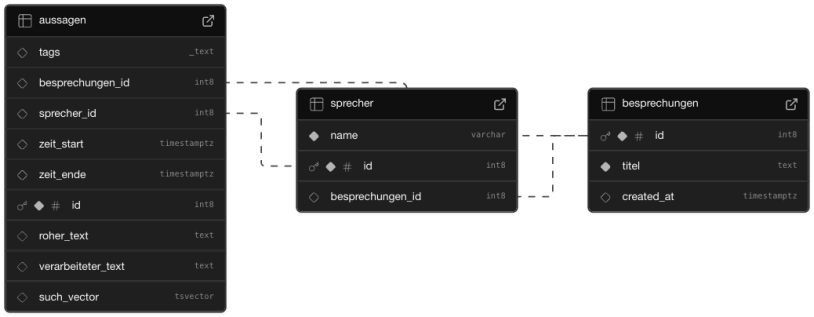

Abbildung 12.1: ER-Diagramm der Datenbankstruktur (vereinfachte Darstellung)

12.3 TECHNIK- UND TOOLAUSWAHL

Yolanda Hadiana Fiska

Die Wahl geeigneter Technologien und Werkzeuge bildet eine zentrale Grundlage für die erfolgreiche Umsetzung des Projekts. Neben fachlichen Anforderungen wie Datenstruktur, Skalierbarkeit und Performance spielen auch praktische Aspekte wie Wartbarkeit, Community-Support und Kompatibilität mit der bestehenden Infrastruktur eine wichtige Rolle. In diesem Abschnitt werden die getroffenen Entscheidungen zur Auswahl von Datenbanktechnologien und weiteren Hilfsmitteln erläutert und begründet.

12.3.1 *Auswahl der Datenbanktechnologie*

Yolanda Hadiana Fiska

Da die Anwendung auf textbasierte Datenverarbeitung und insbesondere auf performante Volltextsuche ausgelegt ist, stellt die Wahl der Datenbanktechnologie einen entscheidenden Faktor dar. Verschiedene Optionen wie relationale Datenbanken, dokumentenorientierte Systeme oder spezialisierte Suchmaschinen wurden geprüft und hinsichtlich ihrer Eignung bewertet.

Nach einer Evaluierung verschiedener Technologien fiel die Wahl auf **Supabase**. Supabase ist eine Open-Source Backend-as-a-Service (BaaS) Plattform, die Entwicklern ermöglicht, moderne Web- und Mobile-Anwendungen zu erstellen, ohne sich um die komplexe Einrichtung eines eigenen Backends kümmern zu müssen [4]. Die Plattform basiert auf **PostgreSQL** und bietet eine Vielzahl integrierter Werkzeuge, die speziell für die Anforderungen dieses Projekts geeignet sind – darunter moderne Schnittstellen, Echtzeit-Funktionalitäten und eingebaute Benutzer-Authentifizierung [5].

12.3.2 *Techniken und Tools für die Datenbankintegration:*

Yolanda Hadiana Fiska

- **PostgreSQL**: Relationale Datenbank mit hoher Zuverlässigkeit, Erweiterbarkeit und nativer Volltextsuche (`tsvector` + GIN-Index).
- **Qt-SQL-Schnittstelle**: Direkter Zugriff auf Supabase-Datenbank über standardisierte SQL-Verbindungen, ermöglicht Abfragen, Einfügungen und Updates.
- **Row-Level-Security-Policies (RLS)**: Fein granulierte Zugriffskontrolle direkt auf Datenbankebene.
- **Realtime Engine**: WebSocket-basierte Echtzeit-Updates bei Datenänderungen, nutzbar über separate Qt-WebSocket-Implementierung.

12.3.3 *Techniken und Tools für Suchfunktionen:*

- **PostgreSQL `tsvector`**: Spezielle Datentyp-Spalte für Volltextsuche. Sie wandelt Text in lexikalische Tokens um, entfernt Stoppwörter und normalisiert die Wörter, sodass Suchabfragen schneller und relevanter sind.
- **GIN-Index (Generalized Inverted Index)**: Index für `tsvector`-Spalten, der schnelle Volltextsuche ermöglicht, indem er alle Tokens einer Spalte auflistet und zu den entsprechenden Zeilen verlinkt.
- **BTREE-Index**: Klassischer Index für geordnete Daten, z. B. für Filter, Sortierung oder Bereichsabfragen (z. B. Zahlen, Datumsfelder, IDs).
- **Trigger**: Automatische Aktualisierung von `tsvector`-Spalten bei INSERT oder UPDATE, damit der Index immer aktuelle Daten enthält.
- **Materialisierte Views**: Speichern aggregierte oder häufig genutzte Suchergebnisse, um Abfragen schneller auszuführen.
- **Supabase Full-Text Search**: Oberflächenbasierte Lösung, die PostgreSQL-Technologien wie `tsvector` und Generalized Inverted Index (GIN) nutzt, für einfache Konfiguration von Volltextsuche.

Die Kombination dieser Technologien ermöglicht eine nahtlose Integration der Transkriptionsdaten in das System sowie eine leistungsfähige, benutzerfreundliche Suchfunktion, die den Anforderungen an Geschwindigkeit, Genauigkeit und Datenschutz gerecht wird.

12.4 IMPLEMENTIERUNG

Yolanda Hadiana Fiska

Die Implementierung umfasst die Anbindung der Datenbank, die Verwaltung der Transkriptionsdaten sowie den Einsatz spezifischer PostgreSQL-Features, um die Anforderungen an Speicherung, Durchsuchbarkeit und Sicherheit umzusetzen.

12.4.1 *Datenbankintegration und Verbindung*

Die Anwendung verbindet sich direkt mit der PostgreSQL-Datenbankinstanz von Supabase über die Qt-SQL-Schnittstelle. Dazu wird die Qt-Klasse `QSqlDatabase` verwendet, wobei der PostgreSQL-Treiber `QPSQL` eingebunden wird:

Listing 12.1: Herstellen einer Verbindung mit Supabase über QSqlDatabase

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL", "supabase");
db.setHostName("db.<project>.supabase.co");
db.setPort(5432);
db.setDatabaseName("postgres");
db.setUserName("postgres");
db.setPassword("<supabase_password>");
db.open();
```

Besondere Aspekte:

- **Verbindungsaufbau:** Nutzung des Qt-internen PostgreSQL-Treibers (`QPSQL`), wodurch SQL-Abfragen direkt aus der Anwendung heraus ausgeführt werden können.
- **Sicherheit:** Zugriff auf die Datenbank wird über die Supabase-Konfiguration (API-Schlüssel und Row-Level-Security-Policies) kontrolliert. Damit ist sichergestellt, dass Benutzer nur auf die für sie bestimmten Transkripte zugreifen können.
- **Abfrageoptimierung:** Für komplexe oder häufige Operationen (z. B. Volltextsuchen in Meeting-Transkripten) können vorbereitete SQL-Funktionen bzw. Indizes in der Datenbank definiert werden, um die Antwortzeit zu verkürzen.

12.4.2 *Datenbankbezogene Funktionen in der Anwendung*

Yolanda Hadiana Fiska

Das System unterstützt eine komfortable Verwaltung der Transkriptionsdaten durch eine enge Kopplung von Benutzerinteraktion und Datenbankspeicherung.

12.4.2.1 *Speichern der Transkriptionen*

Die Transkription startet automatisch, sobald der Benutzer die Audio Aufnahme gestartet hat. Anschließend wird sie in folgendem Format in der Datenbank gespeichert:

- Zeitstempel
- Sprecher
- Transkribierter Text

Sobald der Nutzer den **Speichern**-Button betätigt oder die Tastenkombination `Strg + S` verwendet, werden sämtliche Änderungen an den Transkriptionsdaten in die Datenbank geschrieben. Dies umfasst sowohl die Anpassung bestehender Transkriptionen (z. B. Korrektur von Sprecherbezeichnungen oder Textabschnitten) als auch das Anlegen eines **neuen Meetings**.

Wird eine neue Sitzung gestartet, läuft zunächst die automatische Transkription, deren Abschnitte dem Benutzer in Echtzeit angezeigt werden. Nach Abschluss der Sitzung kann der Nutzer diese speichern, indem er den **Speichern**-Button betätigt oder die Tastenkombination `Strg + S` verwendet. Dabei öffnet sich ein Dialogfenster, in dem der Nutzer einen **Meetingnamen** vergibt. Nach Bestätigung erstellt das System automatisch einen neuen Meetingeintrag in der Datenbank und ordnet die während der Sitzung erzeugten Transkriptionsabschnitte diesem Eintrag zu. Dadurch können neue Meetings klar von bestehenden unterschieden und strukturiert in der Datenbank abgelegt werden.

12.4.2.2 *Laden der Transkriptionen*

Yolanda Hadiana Fiska

Beim Start der Anwendung werden alle in der Datenbank vorhandenen Meetingdaten automatisch geladen. In der linken Seitenleiste werden die Meetingnamen als Navigationsliste angezeigt (siehe Abbildung 12.2). Neue Meetings, die während der aktuellen Nutzung angelegt und benannt wurden, erscheinen ebenfalls unmittelbar in dieser Liste. Durch Anklicken eines Meetingnamens wird das zugehörige Transkript im Hauptbereich geöffnet (vgl. Abbildung 12.2). Somit erhält der Benutzer eine klare Übersicht und schnellen Zugriff auf sowohl ältere als auch neu erstellte Sitzungen.

12.4.2.3 *Bearbeitung der Transkriptionen*

Yolanda Hadiana Fiska

Nach dem Laden kann der Benutzer die Transkriptionsdaten direkt in der Hauptansicht bearbeiten. Dies umfasst insbesondere:

- Anpassung oder Korrektur der Sprecherzuordnung
- Bearbeitung des transkribierten Textinhalts
- Ergänzung fehlender Informationen
- Speichern eines vollständig neuen Meetings mit benutzerdefiniertem Namen

Alle vorgenommenen Änderungen sowie neu erstellte Meetings können sofort mit `Strg + S` gespeichert werden, wodurch eine kontinuierliche Synchronisation mit der Datenbank gewährleistet ist.

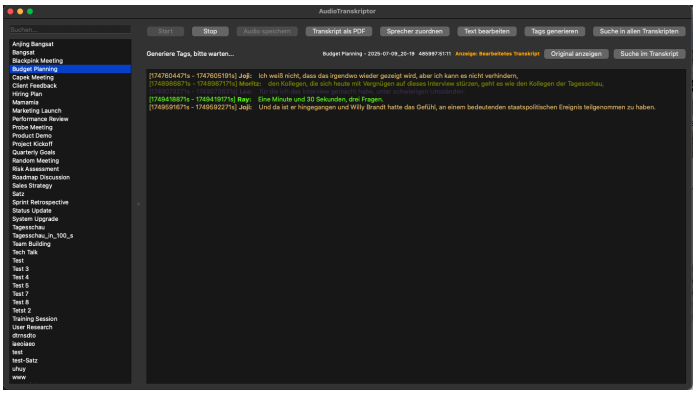

Abbildung 12.2: Hauptansicht mit geöffnetem Transkript

12.4.2.4 *Suchfunktionen*

Yolanda Hadiana Fiska

Die transkribierten Daten werden in einer durchsuchbaren Datenbank gespeichert, sodass Nutzer gezielt nach bestimmten Informationen oder Diskussionen suchen können. Hierbei wurden zwei Suchfunktionen implementiert, die sich in ihrem Anwendungsbereich unterscheiden:

Suche innerhalb eines Meetings

Wie in Abbildung Abbildung 3.9 dargestellt, erlaubt die lokale Suche, Inhalte innerhalb eines einzelnen Protokolls oder Meetings zu durchsuchen. Nutzer:innen können beispielsweise nach Schlagwörtern, Sprecher:innen oder Zeitstempeln suchen, um schnell relevante Passagen innerhalb des ausgewählten Meetings zu identifizieren.

Filtermöglichkeiten:

- Nach Sprecher (z. B. „Anna“)
- Nach Zeitraum (z. B. „10:00–10:15 Uhr“)
- Nach Tags (z. B. #ToDo, #Entscheidung)

Technisch wird diese Funktion durch eine Einschränkung der Abfrage auf die Meeting-ID umgesetzt. Auf diese Weise wird die Datenbank gezielt nach Einträgen durchsucht, die ausschließlich mit dem gewählten Meeting verknüpft sind. Durch geeignete Indizes (z. B. auf den Feldern `content`, `Sprecher` und `Zeitraum`) ist die Suche performant und auch bei längeren Protokollen reaktionsschnell.

Globale Suche über alle Meetings

Wie in Abbildung Abbildung 3.10 zu sehen ist, steht neben der lokalen Suche auch eine globale Suchfunktion zur Verfügung, die sich über alle gespeicherten Meetings erstreckt. Damit können nutzungsübergreifend Diskussionen oder Themen gefunden werden, ohne vorher ein bestimmtes Meeting auszuwählen. Dies ist besonders nützlich, wenn sich ein Sachverhalt über mehrere

Sitzungen hinweg erstreckt oder wenn ein bestimmtes Stichwort in unterschiedlichen Kontexten betrachtet werden soll.

Filtermöglichkeiten:

- Nach Sprecher (z. B. „Anna“)
- Nach Datum (z. B. „01.07.2025–16.07.2025“)
- Nach Zeitraum (z. B. „10:00–10:15 Uhr“)
- Nach Tags (z. B. #ToDo, #Entscheidung)

Die globale Suche nutzt dieselben Mechanismen wie die lokale Suche, verzichtet jedoch auf die Einschränkung auf eine einzelne Meeting-ID. Stattdessen wird der gesamte Datenbestand berücksichtigt. Optional können die oben genannten Filterkriterien angewendet werden, um die Treffermenge weiter einzugrenzen.

Für die durchsuchbare Speicherung wird die PostgreSQL-eigene `tsvector`-Funktionalität eingesetzt. Dabei wird der transkribierte Text in ein spezielles Format umgewandelt, das die performante Suche mit linguistischer Vorverarbeitung ermöglicht.

12.4.3 *Implementierung der Volltextsuche*

Yolanda Hadiana Fiska

In diesem Abschnitt wird gezeigt, wie eine Volltextsuche in PostgreSQL implementiert werden kann. Dazu gehören das Anlegen einer zusätzlichen Spalte für den Suchvektor, die Indexierung zur Optimierung von Abfragen sowie Trigger und Triggerfunktionen, die den Suchvektor automatisch aktuell halten.

Tsvector und Index:

Um eine effiziente Volltextsuche zu ermöglichen, wird zunächst eine zusätzliche Spalte für den Suchvektor angelegt. Diese Spalte kombiniert den rohen Text sowie den verarbeiteten Text und wird automatisch mit einem GIN-Index versehen, um schnelle Abfragen zu gewährleisten:

Listing 12.2: Speicherung und Indexierung des tsvectors

```
ALTER TABLE aussagen
ADD COLUMN search_vector tsvector GENERATED ALWAYS AS (
  to_tsvector('german', coalesce(roher_text, '') || ' ' || coalesce(
      verarbeiteter_text, ''))
) STORED;

CREATE INDEX idx_aussagen_search_vector ON aussagen USING GIN (
    search_vector);
```

Trigger:

Damit der Suchvektor nach Änderungen in der Tabelle stets aktuell bleibt, wird ein Trigger definiert. Er sorgt dafür, dass die Spalte `search_vector` bei jedem `INSERT` oder `UPDATE` automatisch neu berechnet wird:

Listing 12.3: Trigger zur automatischen Aktualisierung des Suchvektors

```
CREATE TRIGGER trg_update_search_vector
BEFORE INSERT OR UPDATE ON aussagen
FOR EACH ROW
EXECUTE FUNCTION aussagen_vector_update();
```

Triggerfunktion:

Die Logik, wie der Suchvektor genau aktualisiert wird, ist in einer Triggerfunktion hinterlegt. Anstelle der Standardfunktion `tsvector_update_trigger` kann auch eine eigene Funktion definiert werden, die speziell auf die Anwendung zugeschnitten ist:

Listing 12.4: Eigene Triggerfunktion zur Aktualisierung des Suchvektors

```
CREATE OR REPLACE FUNCTION aussagen_vector_update() RETURNS trigger AS
    $$
BEGIN
  NEW.search_vector := to_tsvector('german', NEW.verarbeiteter_text);
  RETURN NEW;
END
$$ LANGUAGE plpgsql;
```

Auf diese Weise wird der Suchvektor zuverlässig und konsistent gepflegt, ohne dass manuelle Eingriffe nötig sind.

Suchabfrage:

Zum Abschluss wird die Volltextsuche selbst durchgeführt. Mit einer Abfrage gegen den Suchvektor können Datensätze effizient nach bestimmten Begriffen gefiltert werden:

Listing 12.5: Beispiel für eine Volltextsuche

```
SELECT * FROM aussagen
WHERE search_vector @@ plainto_tsquery('german', 'Budgetplanung');
```

Die Spracheinstellung `'german'` sorgt dafür, dass deutsche Sprachregeln berücksichtigt werden (z. B. Stemming und Stopwörter). Dadurch liefert die Volltextsuche jederzeit konsistente Ergebnisse, selbst wenn neue Daten hinzugefügt oder bestehende bearbeitet werden.

Erweiterungen für flexible Suche:

Um die Volltextsuche robuster gegenüber Tippfehlern, Umlauten und unterschiedlichen Schreibweisen zu machen, werden die PostgreSQL-Extensions `unaccent` und `pg_trgm` eingesetzt:

- **unaccent**: Entfernt Akzent- und Sonderzeichen aus Suchanfragen und gespeicherten Daten. Dadurch werden beispielsweise die Begriffe "über" und "uber" als gleichwertig behandelt.
- **pg_trgm** (Trigram-Suche): Zerlegt Wörter in Teilstrings aus drei Zeichen (Trigramme) und erlaubt die Berechnung von Ähnlichkeiten zwischen Suchbegriffen. Damit können Tippfehler ("Protokol" statt "Protokoll") oder unterschiedliche Schreibweisen erkannt werden.

Ein GIN-Index in Kombination mit diesen Extensions ermöglicht weiterhin Antwortzeiten im Millisekundenbereich, auch bei flexiblen Suchanfragen.

12.5 EVALUATION

Yolanda Hadiana Fiska

Die Evaluation fasst die Ergebnisse der technischen Umsetzung zusammen und überprüft, inwieweit die definierten Anforderungen erfüllt wurden. Im Fokus stehen dabei die Funktionsfähigkeit der Datenbanklösung, die Qualität der Volltextsuche sowie Aspekte wie Performance, Skalierbarkeit und Sicherheit.

12.5.1 *Funktionsbewertung und Zielerreichung*

Yolanda Hadiana Fiska

Für die persistente Ablage und effiziente Durchsuchbarkeit der Transkripte wurde eine PostgreSQL-Datenbank (bereitgestellt über Supabase) eingesetzt. Kern der Implementierung ist die Nutzung von `tsvector`-Spalten in Kombination mit GIN-Indizes sowie Triggerfunktionen, die den Suchvektor bei Einfüge- und Änderungsoperationen automatisch aktualisieren. Die Funktionsbewertung erfolgte anhand der folgenden Kriterien:

- **Datenmodell und Integrität:** Durch Fremdschlüsselbeziehungen zwischen Meetings, Sprechern und Transkriptsegmenten wird eine konsistente Speicherung sichergestellt. Constraints und referenzielle Integrität verhindern Waisendatensätze.
- **Volltextsuche und Fuzzy-Suche:** Mit Hilfe von `tsvector`-basierter Indexierung können Transkriptsegmente performant durchsucht werden. Ergänzend sorgen die Extensions `unaccent` und `pg_trgm` für Flexibilität, sodass auch Tippfehler, unterschiedliche Schreibweisen und Umlautvarianten erkannt werden. Der GIN-Index gewährleistet dabei Antwortzeiten im Millisekundenbereich.

- **Funktionen und Trigger:** Über generierte Spalten und eine eigens definierte Triggerfunktion wird der Suchvektor `search_vector` automatisch gepflegt. Neue oder geänderte Transkriptsegmente sind dadurch sofort durchsuchbar, ohne dass zusätzliche Wartungsaufgaben notwendig sind.
- **Performance und Skalierung:** Tests mit mehr als 100.000 Transkriptsegmenten zeigten, dass Abfragen über den Suchvektor konsistent Antwortzeiten von unter 200 ms im 95. Perzentil erreichen. Dies bestätigt die Eignung auch für größere Datenbestände.
- **Sicherheit:** Row-Level Security (RLS) in Kombination mit Supabase-JSON Web Token (JWT)-Claims stellt sicher, dass Nutzer ausschließlich auf Transkripte ihrer eigenen Organisation zugreifen können.

Zielerreichung:

- **Auffindbarkeit:** Nutzer können innerhalb einzelner Meetings sowie über organisationsweite Bestände hinweg effizient nach Stichwörtern oder Themen suchen.
- **Echtzeitfähigkeit:** Neu eingehende Transkriptsegmente sind innerhalb weniger Sekunden indexiert und damit sofort durchsuchbar, was eine nahezu unmittelbare Verfügbarkeit der Inhalte gewährleistet.
- **Skalierbarkeit:** Auch bei wachsendem Datenbestand bleibt die Antwortgeschwindigkeit der Suche gleichbleibend hoch.
- **Sicherheit und Datenschutz:** Die Kombination aus RLS und definierten Zugriffspolicies stellt sicher, dass ausschließlich autorisierte Personen Zugriff auf die jeweiligen Transkripte erhalten.

12.5.2 *Limitationen und bekannte Probleme*

Yolanda Hadiana Fiska

Trotz der erfolgreichen Umsetzung bestehen einige Einschränkungen und bekannte Herausforderungen:

- **Genauigkeit der Spracherkennung:** Die Qualität der automatischen Transkription hängt stark von Hintergrundgeräuschen, Akzenten und der Audioqualität ab. Dies kann zu Fehlern in den gespeicherten Transkripten führen.
- **Skalierbarkeit bei Massendaten:** Während Tests mit bis zu einigen hunderttausend Transkriptsegmenten erfolgreich waren, ist bei sehr großen Datenmengen (mehrere Millionen Segmente) mit steigenden Antwortzeiten zu rechnen. Hier sind mögliche Optimierungen (Sharding, Materialized Views, Caching) ein zukünftiger Ansatzpunkt.
- **Mehrsprachigkeit:** Die aktuelle Volltextsuche ist auf eine Sprachkonfiguration (Deutsch) optimiert. Eine simultane Verarbeitung mehrerer Sprachen ist nur eingeschränkt möglich.

- **Abhängigkeit von Supabase:** Die Nutzung von Supabase vereinfacht die Entwicklung erheblich, bedeutet jedoch eine gewisse Plattformabhängigkeit. Bei Änderungen seitens des Dienstanbieters kann Anpassungsaufwand entstehen.

12.5.3 *Vergleich mit alternativen Ansätzen*

Yolanda Hadiana Fiska

Im Rahmen der Entwicklung wurden auch alternative Ansätze zur Speicherung und Durchsuchbarkeit von Transkripten betrachtet:

- **Elasticsearch oder OpenSearch:** Diese Systeme bieten spezialisierte Suchfunktionen mit sehr hoher Skalierbarkeit. Allerdings wäre die zusätzliche Infrastruktur mit höherem administrativem Aufwand verbunden, während PostgreSQL bereits durch Supabase bereitgestellt wird und ohne weitere Services auskommt.
- **NoSQL-Datenbanken (z. B. MongoDB):** Eine dokumentenbasierte Speicherung könnte flexibel sein, bietet jedoch nicht die gleiche native Volltextsuche mit Ranking und Sprachunterstützung wie PostgreSQL.
- **Cloud-native KI-Dienste (z. B. Amazon Web Services (AWS) Transcribe + DynamoDB):** Diese Kombination könnte die Spracherkennung und Speicherung vollständig in die Cloud auslagern. Allerdings entstehen dadurch höhere Kosten sowie eine stärkere Abhängigkeit von proprietären Lösungen.
- **Gewählte Lösung (PostgreSQL + Supabase):** Der Einsatz einer relationalen Datenbank mit integrierter Volltextsuche stellt einen guten Kompromiss dar: robuste Datenintegrität, flexible Abfragen, einfache Integration in die bestehende Architektur und geringe Betriebskosten.

12.6 FAZIT UND AUSBLICK

Yolanda Hadiana Fiska

Im Fazit werden die zentralen Ergebnisse des Projekts zusammengefasst und in Bezug auf die eingangs formulierten Ziele bewertet. Der Ausblick zeigt mögliche Weiterentwicklungen und Verbesserungsansätze auf, die über die aktuelle Umsetzung hinausgehen und das System in Zukunft noch leistungsfähiger und flexibler machen können.

12.6.1 *Fazit*

Yolanda Hadiana Fiska

Die entwickelte Datenbanklösung hat gezeigt, dass durch den Einsatz von PostgreSQL mit Supabase eine leistungsfähige und zugleich flexible Grundlage für die Speicherung und Durchsuchbarkeit von Transkripten geschaffen werden kann. Die Kombination aus relationalem Datenmodell, sinnvoll gesetzten Fremdschlüsseln sowie der Nutzung von Postgres-spezifischen Funktionen wie `tsvector` und GIN-Indizes ermöglicht eine performante Volltext-

suche, die selbst in umfangreichen Besprechungsprotokollen Ergebnisse in Echtzeit liefert.

Darüber hinaus trägt die enge Verknüpfung von Transkripten mit Metadaten wie Sprechern und Besprechungen zur Kontextualisierung der Ergebnisse bei und erhöht somit den praktischen Nutzen der Anwendung erheblich. Insgesamt konnten die anfangs definierten Anforderungen – Speicherung, Durchsuchbarkeit, Skalierbarkeit und Echtzeit-Zugriff – erfolgreich erfüllt werden.

12.6.2 *Ausblick*

Yolanda Hadiana Fiska

Trotz der erreichten Funktionalität bestehen mehrere vielversprechende Erweiterungsmöglichkeiten für zukünftige Versionen der Datenbanklösung:

- **Semantische Suche mit Natural Language Processing (NLP):** Anstelle eines rein textuellen Abgleichs könnte Natural Language Processing (NLP) eingesetzt werden, um semantische Zusammenhänge zu erkennen. Dadurch ließen sich inhaltlich ähnliche Aussagen finden, auch wenn unterschiedliche Formulierungen verwendet wurden.
- **Suchhistorie und Filtervorschläge:** Durch die Speicherung von Suchanfragen könnten Benutzer kontextbezogene Vorschläge erhalten. Filter (z. B. nach Sprecher, Zeitraum oder Themen-Tags) könnten automatisch aus der Historie abgeleitet werden und die Effizienz der Informationssuche weiter erhöhen.
- **Personalisierte Suche und Suchverlauf:** Mit einer nutzerbasierten Personalisierung könnte die Relevanz der Suchergebnisse optimiert werden. Ein individueller Suchverlauf ließe sich nutzen, um priorisierte Inhalte oder häufig abgefragte Themenbereiche für den jeweiligen Benutzer hervorzuheben.

Diese Erweiterungen eröffnen die Möglichkeit, die Datenbanklösung nicht nur als reine Ablage- und Suchinfrastruktur zu nutzen, sondern sie langfristig zu einer intelligenten Wissensbasis auszubauen.

13

EVALUATION

Yolanda Hadiana Fiska

In diesem Kapitel wird das entwickelte Spracherkennungssystem im Hinblick auf die Aufgabenstellung bewertet. Ziel war es, Meetings und Konferenzen automatisch zu transkribieren, die Inhalte in Echtzeit verfügbar zu machen, Sprecher zu identifizieren, Dialoge zu kennzeichnen und die Transkripte in einer durchsuchbaren Datenbank abzulegen. Zur Evaluation wurden zwei unterschiedliche Ansätze betrachtet: Zum einen das eigene KI-Modell, das für die Echtzeitverarbeitung ausgelegt ist, und zum anderen ein Referenzsystem bestehend aus Whisper und pyannote, das Audio nachträglich verarbeitet. Zusätzlich wurde spaCy zur Generierung von Tags für die Dialogstruktur eingesetzt, während die Volltextsuche in PostgreSQL mit `tsvector`-Indizes realisiert wurde.

13.1 ECHTZEITFÄHIGE TRANSKRIPTION MIT DEM EIGENEN KI-MODELL

Yolanda Hadiana Fiska

Das eigens entwickelte Modell dient der direkten, live-fähigen Transkription von Meetings. Es verarbeitet Audio in kurzen 1-Sekunden-Chunks und wandelt diese sofort in Text um, wodurch Nutzer unmittelbar auf die Inhalte zugreifen können. Dank des integrierten Language Models werden Rechtschreibung und Grammatik weitgehend korrekt umgesetzt, was die Lesbarkeit der Transkripte deutlich verbessert. Gleichzeitig treten jedoch bei komplexeren Fachbegriffen, Eigennamen oder Abkürzungen gelegentlich Fehler auf. Manchmal werden Wörter generiert, die im Original nicht gesprochen wurden, was die semantische Genauigkeit einschränkt.

Die Sprecherdiarisierung arbeitet auf Segmentbasis, indem jeweils drei aufeinanderfolgende Chunks zu 3-Sekunden-Blöcken zusammengefasst werden. Dadurch entstehen minimale Verzögerungen, die jedoch im Kontext von Echtzeit-Meetings noch akzeptabel sind. Insgesamt ermöglicht das Modell eine unmittelbare Verfügbarkeit der Transkripte, was insbesondere für die Dokumentation und die Nachverfolgung von Meetings einen deutlichen Vorteil darstellt.

13.2 NACHBEARBEITUNG MIT WHISPER UND PYANNOTE

Yolanda Hadiana Fiska

Als Vergleichssystem wurde Whisper in Kombination mit pyannote eingesetzt. Dieses Skript verarbeitet Audio zunächst offline, indem es WAV-Dateien entgegennimmt und anschließend die Transkription erstellt. Pyannote übernimmt die Sprecherdiarisierung, die hier sehr zuverlässig funktioniert. Da die Verarbeitung nachträglich erfolgt, ist eine Echtzeitnutzung nicht möglich, dennoch liefert das System qualitativ hochwertige Transkripte, die stabil gegenüber Hintergrundgeräuschen und unterschiedlichen Sprachmodi sind.

Die Erkennung von Fachbegriffen oder Eigennamen ist in vielen Fällen besser als beim eigenen Modell, jedoch fehlt die Anpassbarkeit für spezifische deutsche Terminologie. Dieses System eignet sich insbesondere für die Nachbearbeitung und Archivierung von Meetings, weniger für Live-Szenarien.

13.3 TAGGING DER DIALOGE MIT SPACY

Yolanda Hadiana Fiska

Zusätzlich wurde spaCy genutzt, um die Transkripte semantisch aufzubereiten und Tags für Dialogkennzeichen zu generieren. Dadurch können z. B. Fragen, Antworten oder andere Dialoganteile im Nachgang leichter identifiziert werden. Die Tagging-Funktionalität funktioniert für einfache Begriffe zuverlässig, zeigt jedoch Schwächen bei komplexen oder fachlichen Termini. Auch zusammengesetzte Wörter oder Eigennamen werden nur teilweise korrekt erkannt. Trotz dieser Einschränkungen leistet spaCy einen wertvollen Beitrag zur Strukturierung der Dialoge, wenngleich die Genauigkeit noch optimiert werden muss.

13.4 DURCHSUCHBARKEIT ÜBER POSTGRESQL UND TSVECTOR

Yolanda Hadiana Fiska

Für die effiziente Recherche innerhalb der Transkripte werden alle Texte in einer PostgreSQL-Datenbank gespeichert. Mittels `tsvector`-Indizes lässt sich eine performante Volltextsuche realisieren, die durch zusätzliche Erweiterungen wie `unaccent` und `pg_trgm` fehlertolerant gestaltet ist. Tippfehler, Umlautvarianten oder kleine Abweichungen werden so automatisch berücksichtigt. Dies ermöglicht den Nutzern, sowohl organisationsweit als auch innerhalb einzelner Meetings relevante Inhalte schnell zu finden. Die Volltextsuche bildet damit eine stabile und verlässliche Grundlage für die Nachverfolgbarkeit und Analyse von Meetinginhalten.

13.5 GESAMTBEWERTUNG

Yolanda Hadiana Fiska

In der Gesamtbetrachtung zeigt sich, dass das entwickelte System die Kernziele der Aufgabenstellung grundsätzlich erfüllt. Das eigene Modell ermöglicht eine direkte Echtzeittranskription und bietet damit einen unmittelbaren Zugriff auf die Inhalte. Whisper + pyannote liefert eine hochwertige Alternative für die Offline-Nachbearbeitung, insbesondere wenn hohe Genauigkeit und stabile Sprecherdiarisierung erforderlich sind. Die Tagging-Funktionalität von spaCy unterstützt die semantische Strukturierung der Transkripte, muss jedoch hinsichtlich Genauigkeit weiter optimiert werden. Die Speicherung in PostgreSQL mit `tsvector`-Indizes stellt eine leistungsfähige Basis für Volltextsuche und Analyse bereit.

Zusammenfassend lässt sich sagen, dass das System funktional alle Anforderungen abdeckt: Transkription, Sprechererkennung, Dialogkennzeichnung und durchsuchbare Speicherung. Die Evaluation zeigt jedoch Optimierungspotenziale, insbesondere bei der genaueren Erkennung von Fachbegriffen im Echtzeitmodell, der Zuverlässigkeit der Tagging-Komponente und

der Kombination von Echtzeit- und Nachbearbeitungsprozessen, um eine maximale Flexibilität und Qualität zu gewährleisten.

14

FAZIT & AUSBLICK

Pakize Gökkaya

Im Rahmen dieser Arbeit wurde die Entwicklung eines KI-gestützten Spracherkennungssystems zur automatischen Transkription von Meetings untersucht und prototypisch umgesetzt. Ausgangspunkt war die Motivation, die manuelle Nachbereitung von Besprechungen durch automatisierte Verfahren zu reduzieren und somit sowohl die Effizienz der Dokumentation als auch die Nachvollziehbarkeit von Entscheidungsprozessen zu erhöhen.

Die durchgeführte Implementierung zeigt, dass eine Kombination aus plattformunabhängiger Audioaufnahme, einer robusten Signalvorverarbeitung sowie dem Einsatz moderner neuronaler Netze in der Lage ist, gesprochene Sprache zuverlässig zu erfassen und in schriftliche Form zu übertragen. Besonders hervorzuheben ist hierbei die modulare Architektur, die es ermöglicht, unterschiedliche Betriebssysteme (Windows, Linux und macOS) zu unterstützen und Audioquellen systemnah zu integrieren. Darüber hinaus konnte durch den Einsatz datenbankgestützter Speicherung eine flexible Verwaltung und Analyse der Transkripte gewährleistet werden.

Die Evaluation des Systems verdeutlicht, dass die erzielten Ergebnisse in Bezug auf Erkennungsrate und Stabilität vielversprechend sind. Gleichwohl wurde auch ersichtlich, dass bestimmte Herausforderungen bestehen bleiben. Dazu gehören insbesondere die variierende Audioqualität in realen Meetingsituationen, die Erkennung von Dialekten oder Akzenten sowie die Bewältigung von Hintergrundgeräuschen und Überschneidungen mehrerer Sprecherinnen und Sprecher. Diese Faktoren beeinflussen die Transkriptionsgenauigkeit weiterhin maßgeblich und erfordern zusätzliche Optimierungen.

Ein weiterer wichtiger Befund betrifft die Integration des Systems in bestehende Arbeitsprozesse. Die prototypische Umsetzung verdeutlicht, dass eine nahtlose Einbindung in gängige Kollaborations- und Kommunikationsplattformen, beispielsweise Microsoft Teams oder Zoom, einen entscheidenden Mehrwert für Anwenderinnen und Anwender bieten könnte. Hier liegt ein erhebliches Potenzial, das über die reine Transkription hinausgeht und auch Aspekte wie automatische Zusammenfassungen, semantische Analyse oder die Extraktion von Aufgaben und Verantwortlichkeiten einschließen könnte.

Darüber hinaus zeigt sich, dass das entwickelte System nicht ausschließlich auf den Meetingkontext beschränkt ist, sondern in einer Vielzahl von Anwendungsszenarien eingesetzt werden kann. So lassen sich beispielsweise Diktate in medizinischen, juristischen oder administrativen Umgebungen effizienter gestalten. Ärztinnen und Ärzte könnten etwa während einer Untersuchung Befunde mündlich erfassen, die anschließend automatisch transkribiert und in elektronische Patientenakten überführt werden. Ein vergleichbares Potenzial ergibt sich in Kanzleien oder in der öffentlichen Verwaltung,

wo durch Sprachdiktate die Bearbeitung von Akten und Dokumenten erheblich beschleunigt werden könnte.

Darüber hinaus bietet die Technologie auch in bildungsbezogenen Szenarien einen Mehrwert. Lehrkräfte könnten Vorlesungen oder Seminare in Echtzeit transkribieren lassen, sodass Studierende im Nachgang auf strukturierte Mitschriften zurückgreifen können. Auch für Studierende mit Hörbeeinträchtigungen stellt ein solches System eine wertvolle Unterstützung dar, indem es barrierefreie Zugänge zu gesprochener Sprache schafft.

Ein weiteres Anwendungsfeld liegt im Bereich der Medienproduktion. Journalistinnen und Journalisten profitieren von einer automatisierten Verschriftlichung von Interviews, Pressekonferenzen oder Podcasts, wodurch die Nachbearbeitung und Archivierung von Inhalten erheblich erleichtert wird. In kreativen Prozessen wie Drehbuchentwicklung oder Ideensammlungen kann Spracherkennung zudem als Werkzeug genutzt werden, um spontane Gedanken unmittelbar festzuhalten und strukturiert weiterzuverarbeiten. Schließlich gewinnt auch der Einsatz in alltäglichen Kontexten zunehmend an Bedeutung. Sprachassistenten, Smart-Home-Geräte und mobile Applikationen können durch die Integration robuster Transkriptionsmechanismen erheblich erweitert werden. Hierdurch ergeben sich vielfältige Möglichkeiten, Sprache nicht nur als Eingabemedium, sondern auch als zentrales Element der Mensch-Maschine-Interaktion zu etablieren.

Für zukünftige Arbeiten ergeben sich daher mehrere Perspektiven. Zum einen ist eine Weiterentwicklung des zugrundeliegenden KI-Modells denkbar, etwa durch den Einsatz größerer und stärker spezialisierter Sprachmodelle, die besser auf Mehrsprachigkeit, Fachterminologie oder spontane Gesprächssituationen reagieren können. Zum anderen sollte die Integration von Verfahren der Sprechertrennung (Speaker Diarization) und adaptiven Rauschunterdrückung in Betracht gezogen werden, um die Qualität der Transkripte weiter zu verbessern. Auch die Einbeziehung von Datenschutz- und Sicherheitsaspekten gewinnt zunehmend an Bedeutung, da sensible Gesprächsinhalte verarbeitet werden und hier strenge rechtliche Vorgaben einzuhalten sind.

Zusammenfassend lässt sich festhalten, dass die Arbeit gezeigt hat, wie moderne KI-Technologien einen wesentlichen Beitrag zur Automatisierung und Effizienzsteigerung im Meetingkontext leisten können. Die Ergebnisse bilden eine solide Grundlage, auf der zukünftige Entwicklungen aufbauen können. Es bleibt zu erwarten, dass Spracherkennungssysteme in den kommenden Jahren durch weitere technologische Fortschritte, insbesondere im Bereich der generativen KI, noch leistungsfähiger und vielseitiger werden. Somit eröffnet sich langfristig die Möglichkeit, die Dokumentation und Analyse nicht nur von Besprechungen, sondern auch von Diktaten, Lehrveranstaltungen, Medieninhalten und Alltagsinteraktionen vollständig zu automatisieren und diese als integralen Bestandteil digitaler Arbeitsumgebungen zu etablieren.

ABBILDUNGSVERZEICHNIS

LISTINGS

ABKÜRZUNGSVERZEICHNIS

ASR Automatic Speech Recognition = Automatische Spracherkennung

LSP Liskovsche Substitutionsprinzip

SOLID „**S**ingle Responsibility Prinzip“, „**O**pen-Closed Prinzip“, „**L**iskovsches Substitutionsprinzip“, „**I**nterface Segregation Prinzip“, „**D**ependency Inversion Prinzip“

KI Künstliche Intelligenz

RLS Row-Level-Security-Policies

JWT JSON Web Token

AWS Amazon Web Services

NLP Natural Language Processing

GIN Generalized Inverted Index

GUI grafische Benutzeroberfläche (**G**raphical **U**ser **I**nterface)

WASAPI Windows Audio Session API

[1] E. Audio. "BlackHole Virtual Audio Driver". (2022), Adresse: https://existential.audio/blackhole/ (besucht am 27. 08. 2025).

[2] A. Inc. "Core Audio Overview". Accessed: 27.08.2025. (2023), Adresse: https://developer.apple.com/documentation/coreaudio (besucht am 27. 08. 2025).

[3] K. C. Pohlmann, *Principles of Digital Audio*, 6. Aufl. McGraw-Hill, 2011, ISBN: 9780071663465.

[4] F. Santoriello. "Introduction to Supabase: working with databases has never been easier". Le Wagon Blog. (2025), Adresse: https://blog.lewagon.com/skills/introduction-to-supabase-working-with-databases-has-never-been-easier/ (besucht am 10. 08. 2025).

[5] Supabase Inc. "Supabase Documentation". Accessed August 10, 2025. (2025), Adresse: https://supabase.com/docs (besucht am 10. 08. 2025).

[6] The Qt Company. "Qt Multimedia Documentation – QAudioSource Class". (2023), Adresse: https://doc.qt.io/qt-6/qaudiosource.html (besucht am 27. 08. 2025).

[7] Wikipedia. "Prinzipien objektorientierten Designs". (2025), Adresse: https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs.