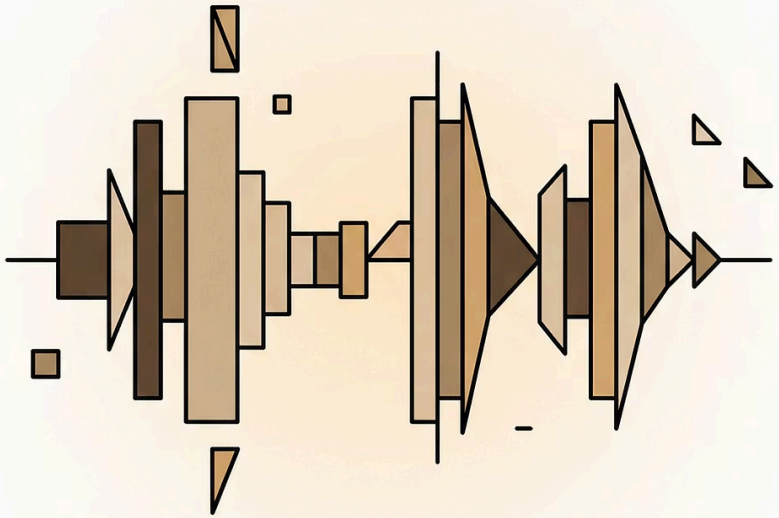


AUDIO WAVEFORM VISUALIZATION

Guidetranslator V.8 — Optimierungsbericht

Ergebnisse der systematischen Tests durch Alexander und Claude mit priorisierten Handlungsempfehlungen für die nächste Iteration.

(Update: 27. Februar 17.323

7 FINDINGS

PRIORISIERT

Kapitel 1

Übersicht der Findings

Die Tests der V.8 haben sieben konkrete Optimierungspotenziale aufgedeckt — von kritischen Netzwerk- und Speicherproblemen bis hin zu langfristigen Architekturverbesserungen. Jedes Finding ist nach Schweregrad und Auswirkung auf Endnutzer priorisiert.

Kritisch

STT-Timeout, Memory-Limit

Hoch

Request-Dedup, Segmentierung

Mittel

TTS-Cancel, Netzwerk-UI

Langfristig

AudioWorklet-Migration

🧑‍💼 **Für Ulrich:** Stell dir vor, die App ist wie ein Dolmetscher, der gleichzeitig zuhört, übersetzt und spricht. Wir haben 7 Stellen gefunden, an denen dieser Dolmetscher ins Stocken geraten kann — und haben für jede Stelle eine Lösung parat. Die schlimmsten Probleme zuerst.

PRIORITÄT 1 — KRITISCH

Google Cloud STT: Kein Timeout + inkrementelles Audio-Problem

Problem

Der `fetch()`-Aufruf zum Google Speech-to-Text-Service hat **keinen konfigurierten Timeout**. Bei Netzwerkproblemen hängt die App mindestens 30 Sekunden lang, ohne dem Nutzer Feedback zu geben. Gleichzeitig wird bei jedem 3-Sekunden-Polling-Intervall der **gesamte Audio-Buffer erneut übertragen** — nicht nur die neuen Daten.

Bei 30 Sekunden Sprechzeit bedeutet das: **~7 MB pro einzelnem Request**. Das potenziert sich bei instabilen Verbindungen zu einem massiven Bandbreitenproblem.

Fix

- **5s AbortController-Timeout** auf allen STT-Fetch-Calls implementieren
- **Inkrementelle Audio-Chunks** senden: Nur neue Daten seit dem letzten erfolgreichen Request übertragen
- Retry-Logik mit exponentiellem Backoff bei Timeout-Fehlern

🧑‍💼 **Für Ulrich:** Aktuell schickt die App bei jeder Anfrage an Google die *gesamte* bisherige Aufnahme nochmal mit — wie wenn du bei jedem neuen Satz das ganze Gespräch nochmal erzählen müsstest. Der Fix: Nur den neuen Teil schicken. Und wenn Google nicht antwortet, nach 5 Sekunden abbrechen statt ewig zu warten.

PRIORITÄT 2 — KRITISCH

Memory-Limit für Audio-Buffer

Zwischen erkannten Satzgrenzen wächst das `audioChunks[]`-Array **unbegrenzt**. Bei einer Abtastrate von 48 kHz (Standard auf iOS-Geräten) und einer Minute ohne erkanntes Satzzeichen akkumulieren sich rund **45 MB Audiodaten im Arbeitsspeicher**.

48kHz	45MB	OOM
iOS Samplerate	**1 Min. Buffer**	**Risiko**
Höhere Rate = größere Buffer	Ohne Satzerkennung	Out-of-Memory bei langen Sessions

Warum das kritisch ist

Bei langen Führungen oder Vorträgen (15+ Minuten) ohne klare Satzgrenzen — etwa bei fließendem Redefluss oder fehlender Interpunktionserkennung — steigt das Risiko eines **Out-of-Memory-Crashes** auf mobilen Geräten erheblich.

Empfohlener Fix

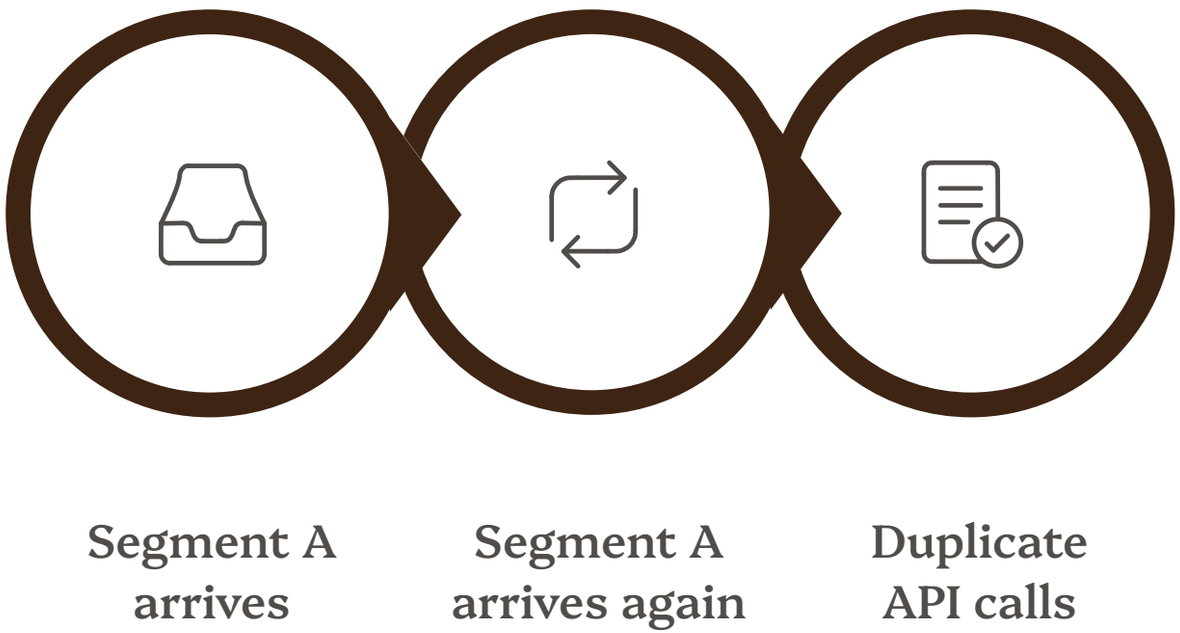
Harter Buffer-Limit von **10 Sekunden Audio (~2 MB max)**, unabhängig von der Satzerkennung. Ältere Chunks werden verworfen, sobald das Limit erreicht ist. Damit bleibt der Speicherverbrauch konstant und vorhersagbar.

> 🧑‍💼 **Für Ulrich:** Die App speichert alles, was gesprochen wird, im Zwischenspeicher des Handys — und vergisst nichts davon. Bei langen Touren kann das den Speicher sprengen und die App abstürzen lassen. Lösung: Nur die letzten 10 Sekunden merken, den Rest löschen.

PRIORITÄT 3 — HOCH

Request-Deduplication für Übersetzungen

Wenn zwei identische Textsegmente eintreffen, **bevor** der erste API-Call zurückkehrt, werden aktuell **zwei separate Übersetzungs-Requests** ausgelöst — eine klassische Race Condition am Translation-Cache.



Der Fix nutzt eine **In-flight Request Map**: Wenn ein identischer Request bereits läuft, wird kein zweiter gestartet. Stattdessen wartet der Duplikat-Request auf das gleiche Promise. Das spart API-Kosten und reduziert die Latenz bei schneller Satzerkennung signifikant.

> 🧑‍💼 **Für Ulrich:** Manchmal erkennt die App den gleichen Satz zweimal kurz hintereinander und schickt ihn doppelt zur Übersetzung — doppelte Kosten, kein Mehrwert. Der Fix sorgt dafür, dass die App bei Dopplern auf die erste Antwort wartet, statt nochmal zu fragen.

PRIORITÄT 4 — HOCH

Live Session: Satz-Segmentierung fehlt

Aktuelles Verhalten

Der `useLiveSession`-Hook profitiert **nicht** von der neuen Satzerkennungslogik. Stattdessen wird der gesamte Final-Result-Text auf einmal an die Übersetzungs-API gesendet — ohne Aufteilung in einzelne Sätze.

Das führt zu:

- Längeren Wartezeiten für Nutzer
- Größeren Übersetzungs-Payloads
- Inkonsistenz zwischen Normal- und Live-Modus

Empfohlener Fix

Die bereits implementierte `detectSentenceBoundary()`-Funktion muss auch im Live-Session-Handler eingebunden werden. Dadurch werden erkannte Sätze **inkrementell und einzeln** zur Übersetzung geschickt — genau wie im Standardmodus.

Erwarteter Effekt: **50–70% schnellere erste Übersetzungsanzeige** bei Live-Sessions, da kürzere Segmente schneller verarbeitet werden.

🧑‍💼 **Für Ulrich:** Wir haben eine clevere Funktion eingebaut, die erkennt, wann ein Satz zu Ende ist — aber im Live-Modus wird sie nicht genutzt. Der Fix schaltet sie auch dort ein, damit Übersetzungen schneller und häppchenweise erscheinen statt als großer Block.

PRIORITÄT 5 — MITTEL

TTS-Queue: Fehlerhafte cancel()-Logik

Die Text-to-Speech-Ausgabe hat ein subtiles Timing-Problem: `window.speechSynthesis.cancel()` wird **bei jedem neuen Segment aufgerufen** — unabhängig davon, ob aktuell etwas abgespielt wird.

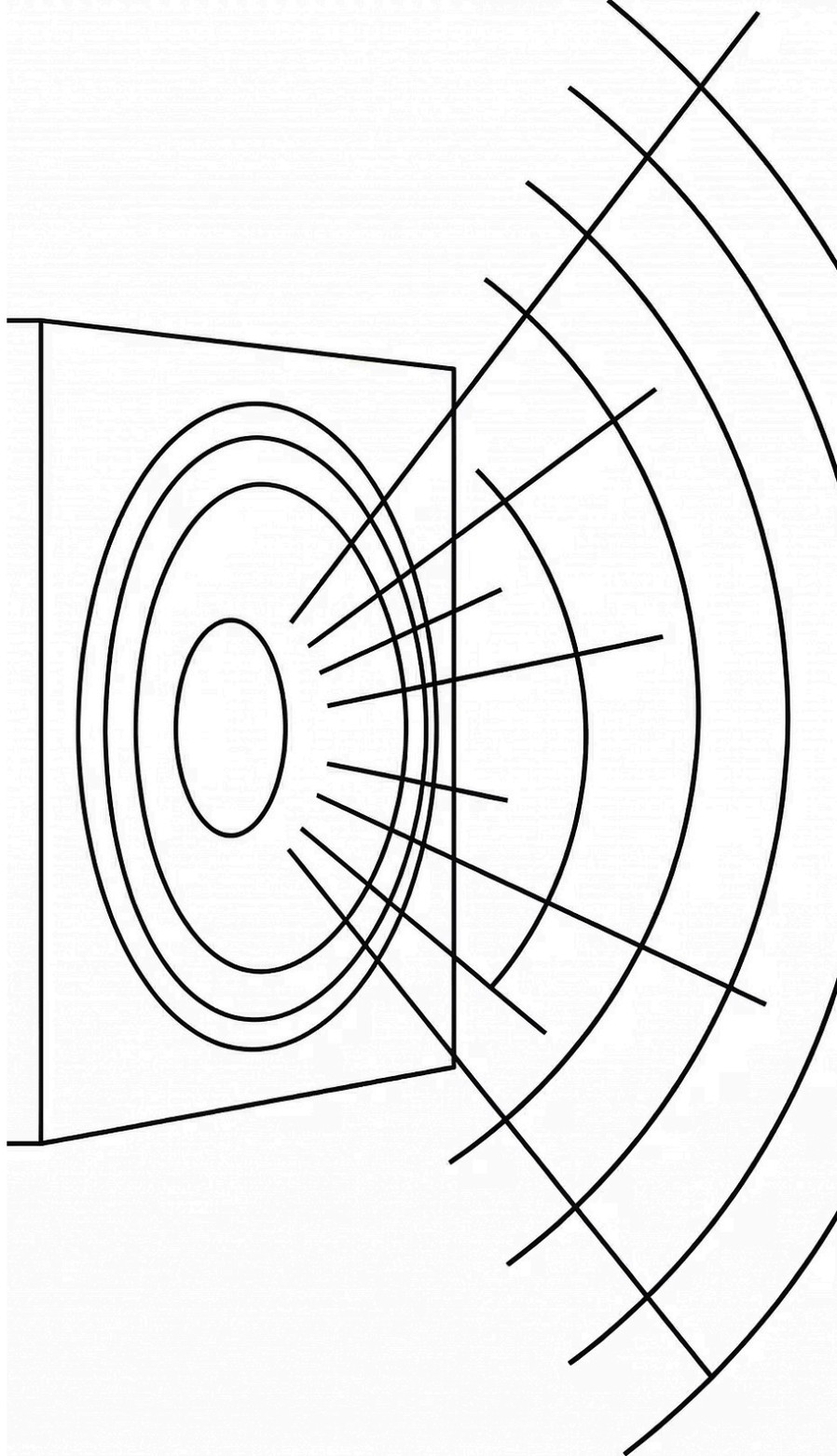
Symptom

Im Satz-Modus mit schnellen Übersetzungen kann `cancel()` vorherige Sätze abbrechen, bevor sie fertig gesprochen sind. Der Nutzer hört abgehackte, unvollständige Ausgaben.

Fix

Nur `cancel()` aufrufen, wenn `speechSynthesis.speaking === true`. Ansonsten das neue Segment in eine Queue einreihen und sequenziell abspielen lassen.

🧑‍💼 **Für Ulrich:** Stell dir vor, die App liest dir einen Satz vor — und mitten drin kommt der nächste Satz und unterbricht den ersten. So klingt es aktuell manchmal. Der Fix sorgt dafür, dass jeder Satz zu Ende gesprochen wird, bevor der nächste beginnt.



PRIORITÄT 6 — MITTEL

Netzwerkausfall während Live Session

Problem

Wenn die Netzwerkverbindung während einer aktiven Live Session abbricht, erhält der Nutzer **keinerlei visuelles Feedback**. Die App erscheint eingefroren — ohne Erklärung, ohne Handlungshinweis. Besonders bei Outdoor-Führungen mit wechselnder Mobilfunkabdeckung ist das ein häufiges Szenario.

Der Nutzer weiß nicht, ob die App noch arbeitet, ob er neu starten muss, oder ob die Verbindung gleich zurückkommt.

1

Connection-Status Monitoring

`navigator.onLine` + `fetch`-Healthcheck alle 5 Sekunden während aktiver Session

2

UI-Indikator

Toast-Banner mit „Verbindung unterbrochen — versuche erneut..." und Spinner-Animation

3

Auto-Reconnect

Automatisches Wiederaufnehmen der Session, sobald Netzwerk verfügbar — ohne manuellen Neustart

🧑‍💼 **Für Ulrich:** Wenn beim Benutzen das Internet kurz weg ist, merkt die App das aktuell nicht — sie friert einfach ein. Der Fix zeigt dem Nutzer an „Hey, Verbindung weg!" und verbindet sich automatisch wieder, sobald Netz da ist.

PRIORITÄT 7 — LANGFRISTIG

Migration zu AudioWorklet

Der aktuell verwendete `ScriptProcessorNode` ist **offiziell deprecated** durch das W3C. Er läuft auf dem Main Thread und blockiert bei hoher Last die UI. Die moderne Alternative — `AudioWorklet` — verarbeitet Audio in einem separaten Rendering-Thread.

ScriptProcessorNode	AudioWorklet
Main Thread • deprecated • blockiert UI bei Last • einfaches Setup	Separater Thread • **20–30% weniger CPU** • benötigt Worker-File • zukunftssicher

Warum langfristig?

Die Migration erfordert ein **separates Worker-File**, Anpassungen am Build-System und gründliches Cross-Browser-Testing. Der Aufwand ist moderat, der Benefit vor allem auf älteren Mobilgeräten mit begrenzter CPU-Leistung spürbar. Empfehlung: In Sprint 3 oder 4 einplanen, nicht als Hotfix.

> 🧑‍💼 **Für Ulrich:** Die Technik, mit der die App Ton aufnimmt, ist veraltet und frisst unnötig viel Leistung. Es gibt eine neuere, effizientere Methode — aber der Umbau dauert etwas. Das planen wir für später, weil es nicht brennt, aber langfristig wichtig ist.

Zusammenfassung

Roadmap & Empfohlene Reihenfolge

Basierend auf Schweregrad, Nutzerimpact und Implementierungsaufwand empfehlen wir folgende Umsetzungsreihenfolge. Die kritischen Findings 1–2 sollten **sofort im nächsten Sprint** adressiert werden.

01

Sprint 1: STT-Timeout + Memory-Limit

AbortController-Timeout, inkrementelle Chunks, harter Buffer-Limit — eliminiert Crash-Risiken

02

Sprint 1: Request-Deduplication

In-flight Map implementieren — geringe Komplexität, sofortige API-Kosteneinsparung

03

Sprint 2: Live-Segmentierung + TTS-Queue

detectSentenceBoundary() in Live Session einbauen, cancel()-Logik korrigieren

04

Sprint 2: Netzwerk-UI-Feedback

Connection-Status-Monitoring und Auto-Reconnect für Live Sessions

05

Sprint 3–4: AudioWorklet-Migration

ScriptProcessorNode ablösen — 20–30% CPU-Einsparung auf Mobilgeräten

🧑‍💼 **Für Ulrich:** Die wichtigsten Baustellen (App hängt, Speicher läuft voll) werden sofort behoben. Danach kommen die Verfeinerungen (schnellere Übersetzungen, bessere Tonausgabe). Den großen Technik-Umbau machen wir am Ende, wenn alles andere stabil läuft.

Nächste Schritte

1

Sprint Planning

Findings 1–3 in den nächsten Sprint aufnehmen. Geschätzter Aufwand: **2–3 Entwicklertage** für alle drei Fixes gemeinsam.

2

Testabdeckung

Für jeden Fix gezielte **Regressionstests** schreiben — besonders Netzwerk-Timeouts und Memory-Limits sind schwer manuell zu testen.

3

Monitoring

Nach Deployment: **Error-Tracking** für STT-Timeouts und OOM-Events einrichten, um den Erfolg der Fixes zu messen.

> Die V.8 ist funktional solide — mit diesen 7 gezielten Optimierungen wird sie auch unter Realbedingungen stabil und performant.

🧑‍💼 **Für Ulrich:** Kurz gesagt: Wir wissen genau, was zu tun ist. Die wichtigsten Reparaturen dauern 2–3 Tage, danach testen wir gründlich und überwachen, ob alles hält. Die App wird dadurch deutlich stabiler und schneller.