

Bharat AI-SoC Student Challenge

Problem Statement 4:

Real-Time On-Device Speech-to-Speech Translation using SME2 and/or NEON on Arm CPU

Pratik Raj, Meet Rana, Praval Gupta

Department of Electrical Engineering

Indian Institute of Technology (IIT) Tirupati

Tirupati, Andhra Pradesh, India

Email: {ee23b042, ee23b070, ee23b043}@iittp.ac.in

20 February 2026

Contents

1	Introduction	2
1.1	Objective	2
1.2	Project Description	2
2	System Overview	2
2.1	Architecture	2
2.2	Key Software Components	3
2.2.1	MainActivity	3
3	Methodology	4
3.1	Detailed Pipeline Logic	4
3.1.1	Audio Capture and Pre-processing	4
3.1.2	Automatic Speech Recognition (Whisper)	4
3.1.3	Text-to-Text Translation (Gemma)	4
3.1.4	Text-to-Speech (MMS-TTS)	5
3.2	Model Preparation and Deployment	6
4	Hardware and Resource Usage	7
4.1	Device Characteristics	7
4.2	Measured Utilization and Latency	7
5	Optimization Techniques	9
5.1	Model-Level	9
5.2	System-Level Optimisations	9
6	Results and Discussion	10
6.1	Latency and Quality	10
6.2	Trade-offs and Limitations	10

1 Introduction

Cloud-based translation services offer high-quality speech translation but depend on connectivity, raise privacy concerns, and add latency. Recent progress in compact speech and language models enables complete speech-to-speech translation (S2S) pipelines to run on-device on consumer smartphones. This work implements such a pipeline on Android using Whisper for automatic speech recognition (ASR), a compact Gemma-family model for text-to-text translation, and Meta’s MMS-TTS models(.onnx) for text-to-speech (TTS).

1.1 Objective

Build a fully local, real-time speech-to-speech translation system optimized for Arm-based CPUs, leveraging SME2 where available, must perform speech recognition, LLM-based translation or semantic rewriting, and speech synthesis entirely on-device, meeting mobile latency, power, and thermal constraints.

1.2 Project Description

To design and deploy a real-time, on-device speech-to-speech translation pipeline running on a smartphone with an Arm-powered CPU(vivo X300). The system captures continuous spoken audio in Language A, performs:

- On-device speech-to-text (STT),
- LLM-based translation or semantic rewriting, and
- Text-to-speech (TTS) synthesis,

to produce natural, fluent spoken output in Language B. All inference run locally with no cloud dependency, demonstrating efficient use of Arm CPU acceleration and mobile-friendly optimizations using SME2.

The final application records user speech, transcribes it with Whisper-tiny int8, translates text with a Gemma-family model, and synthesizes the translation using MMS-TTS, all locally on the device.

2 System Overview

2.1 Architecture

The speech-to-speech translation pipeline is organized as follows:

1. **Audio Input:** Microphone capture using `AudioRecord`, with Voice Activity Detection (VAD).
2. **Automatic Speech Recognition (ASR):** A Whisper model exported for on-device mobile inference.
3. **Translation:** A compact Gemma-family model operating in translation-style prompting mode.
4. **Text-to-Speech (TTS):** Meta MMS-TTS per-language models executed via `sherpa-onnx OfflineTts`.
5. **Playback:** Synthesized audio output through `AudioTrack`.

All models are executed entirely on-device. The ASR and LLM components are invoked through a native shared library (`translator_native.so`) to enable SME2, while MMS-TTS is accessed through the `sherpa-onnx` Android bindings. Conceptually, the system can be represented as:

Microphone → Whisper ASR → Gemma MT → MMS-TTS → Speaker.

2.2 Key Software Components

2.2.1 MainActivity

MainActivity manages the user interface, lifecycle events, and language selection. A mapping connects human-readable language names to two-letter ISO-style codes, for example:

English → "en", Hindi → "hi", French → "fr", Spanish → "es",
German → "de", Tamil → "ta", Arabic → "ar".

The selected codes are written to `PipelineManager.sourceLanguageCode` and `PipelineManager.targetLanguageCode`. A single button toggles recording via `startRecording()` and `stopRecording()`, while a status label and coloured indicator reflect the current state (initialising, listening, translating, ready).

PipelineManager

PipelineManager orchestrates the end-to-end speech-to-speech (S2S) pipeline. Its responsibilities include:

Initialising and releasing JNI-wrapped Whisper and Gemma models via:

```
nativeWhisperInit / nativeWhisperTranscribe / nativeWhisperFree
nativeGemmaInit   / nativeGemmaTranslate   / nativeGemmaFree
```

Maintaining a mapping from two-letter language codes to MMS three-letter codes, e.g.

"en" → "eng", "hi" → "hin", "fr" → "fra",
"es" → "spa", "de" → "deu", "ta" → "tam", "ar" → "ara".

which are used to select the correct MMS-TTS model.

Managing audio capture, sending recorded PCM to Whisper for ASR, constructing a translation prompt and invoking the native Gemma translation function.

Streaming translation tokens back to the UI as they are generated, and, once a sentence or full translation is ready, calling `MmsTtsManager.generateSamples(...)` and playing the returned audio samples via `AudioTrack`.

MmsTtsManager

MmsTtsManager wraps the `sherpa-onnx` `OfflineTts` interface. Its responsibilities include:

Receiving a base `modelDir`, for example:

/storage/emulated/0/Android/data/com.example.speechtranslator/files/mms_tts.

On invocation of `generateSamples(text, mmsCode)`, dynamically loading the corresponding language model if the MMS code has changed, by checking for: `modelDir/mmsCode/model.onnx` and `modelDir/mmsCode/tokens.txt`.

Constructing an `OfflineTtsConfig` with an `OfflineTtsVitsModelConfig`, instantiating `OfflineTts`, and calling `OfflineTts.generate(...)` to synthesise speech for the given text.

Returning a `FloatArray` containing the generated PCM audio samples, which `PipelineManager` then feeds into an `AudioTrack` for playback.

Native Layer

The native layer consists of a C/C++ shared library that wraps the Whisper and llm runtimes. It exposes only initialisation, inference, and teardown functions to the Android layer (for ASR: `nativeWhisperInit/Transcribe/Free`, for MT: `nativeGemmaInit/Translate/Free`). All models are stored in quantised form and configured for efficient execution on mobile CPU cores, reducing memory footprint and inference latency and enabling fully offline, on-device operation.

3 Methodology

3.1 Detailed Pipeline Logic

3.1.1 Audio Capture and Pre-processing

When the user taps the *Record* button, `MainActivity` performs the following steps:

- Instantiates and starts an `AudioRecord` object configured for 16 kHz, mono, PCM encoding.
- Buffers incoming audio into fixed-size chunks (e.g. 500 ms).
- Optionally applies Voice Activity Detection (VAD), such as a Silero VAD model in ONNX format, or a simple RMS-based energy threshold to detect speech segments.

When VAD and/or energy thresholds indicate an end-of-utterance condition, the accumulated PCM samples are forwarded to `PipelineManager` for speech recognition.

3.1.2 Automatic Speech Recognition (Whisper)

The Whisper model is initialized once during application startup via:

```
nativeWhisperInit(whisperPath, numThreads)
```

Here, `whisperPath` refers to a quantized multilingual Whisper checkpoint stored within the application’s file directory.

For each detected utterance:

1. The buffered audio samples are converted into a `FloatArray`.
2. `nativeWhisperTranscribe(pcm, sourceLanguageCode)` is invoked.
3. Internally, Whisper computes log-Mel spectrogram features and performs transformer-based decoding to generate token sequences, which are then converted into text.
4. The resulting transcription string is returned to the Kotlin layer and displayed in the UI.

Small model variants (e.g. `tiny` or `small`) combined with INT8 quantization are used to balance word error rate, memory usage, and inference latency on mobile CPUs.

3.1.3 Text-to-Text Translation (Gemma)

Instead of a dedicated neural machine translation model, a compact Gemma-family model is employed in translation mode using prompt-based conditioning.

For each transcription:

1. `PipelineManager` constructs a structured prompt of the form:

Translate from <src> to <tgt>: "<transcript>"

2. The prompt is passed to the native runtime:

```
nativeGemmaTranslate(prompt, tokenCallback)
```

3. The LLM generates target-language tokens autoregressively. Each generated token is streamed back to the Kotlin layer via `tokenCallback`.
4. `PipelineManager` appends tokens to a string buffer and updates the UI in real time to display the progressively generated translation.

The translation model is quantized to 4–8 bit precision and configured with a context window sufficient for typical spoken sentences, ensuring efficient execution on mobile hardware.

3.1.4 Text-to-Speech (MMS-TTS)

The text-to-speech stage proceeds as follows:

1. `PipelineManager` converts the two-letter target language code (e.g. "hi") into the corresponding MMS three-letter code using the `LANG_TO_MMS` mapping (e.g. "hin").
2. It invokes:

```
val samples = ttsManager.generateSamples(translation, mmsCode)
```

3. `MmsTtsManager` performs:

- If `mmsCode` differs from the currently loaded model, the previous model instance is released and the new model is loaded from:

```
modelDir/mmsCode/model.onnx, modelDir/mmsCode/tokens.txt.
```

- Construction of an `OfflineTtsConfig` object with an `OfflineTtsVitsModelConfig`.
- Instantiation of `OfflineTts` and invocation of:

```
generate(text, sid=0, speed=0.5)
```

- The method returns synthesized PCM samples as a floating-point array.

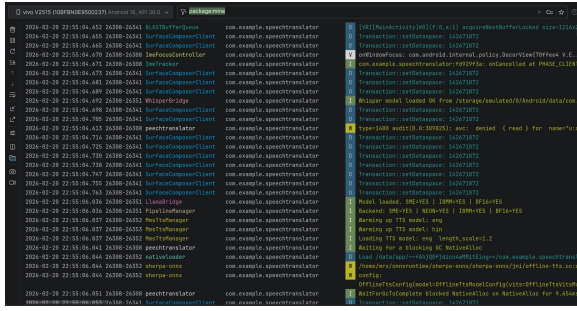
4. `PipelineManager` creates an `AudioTrack` instance configured with:

- Sample rate of approximately 22,050 Hz, mono, PCM float encoding.
- Buffer size defined as:

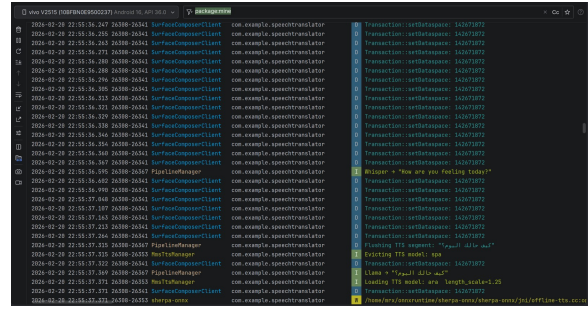
```
max(getMinBufferSize, 4 × samples.length).
```

5. The synthesized samples are written to `AudioTrack` and played through the device speaker.

MMS-TTS models are stored locally under per-language subdirectories (e.g. `eng`, `hin`, `fra`) within the designated application storage path.



(a) App Init



(b) Whisper to Gemma

Figure 1: Images for the Initialization of the app and conversion from Whisper to Gemma

3.2 Model Preparation and Deployment

Whisper A multilingual Whisper checkpoint is exported to a mobile-compatible format (e.g. ONNX or ggml) and quantised to INT8 or mixed precision using standard post-training quantisation workflows. The resulting ggml model file is bundled with the application or downloaded at runtime and stored within the application’s private file directory at

`/storage/emulated/0/Android/data/com.example.speechtranslator/files/models/`

from where the native ASR runtime loads it during `nativeWhisperInit`.

Gemma A compact Gemma-family model is quantised to 4–8-bit precision and integrated through a C++ runtime optimised for on-device generative inference. The ggml model file is stored alongside the Whisper model at

`/storage/emulated/0/Android/data/com.example.speechtranslator/files/models/`

Only a minimal JNI interface—covering initialisation, token generation (streamed via a callback), and teardown—is exposed to the Android layer, reducing overhead and simplifying integration with `PipelineManager`.

MMS-TTS Individual MMS-TTS language models are downloaded and unpacked such that each supported target language has a directory containing:

- `model.onnx`
- `tokens.txt`

All three model types reside under the application’s private external storage directory:

```
/storage/emulated/0/Android/data/com.example.speechtranslator/files/
|-- models/
|   |-- whisper.ggml      (Whisper ASR)
|   |-- Gemma.ggml       (Gemma translation)
|
|-- mms_tts/
|   |-- eng/              (English TTS)
|   |   |-- model.onnx
|   |   |-- tokens.txt
|   |
|   |-- hin/              (Hindi TTS)
|   |   |-- model.onnx
```

```
|  \-- tokens.txt
|
|-- fra/, spa/, deu/, tam/, ara/ (other languages)
```

This modular directory layout enables `MmsTtsManager` to dynamically load and unload language-specific TTS models at runtime based on the MMS code requested by `PipelineManager`.

4 Hardware and Resource Usage

4.1 Device Characteristics

All experiments were conducted on a **vivo X300** smartphone. Table 1 summaries the hardware characteristics relevant to on-device speech and language model inference.

Table 1: Test Device Specifications (Vivo X300)

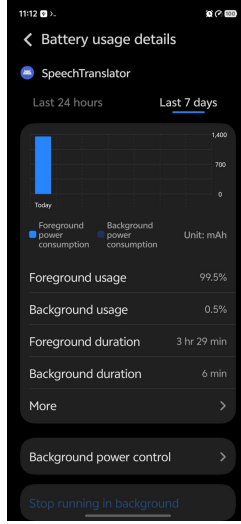
Component	Specification
Model	Vivo X300
Operating System	Android 16 (OriginOS 6)
Chipset	MediaTek Dimensity 9500 (3 nm)
CPU	Octa-core 1×4.21 GHz + 3×3.5 GHz + 4×2.7 GHz
GPU	ARM G1-Ultra
RAM	12+12 GB (Internal: 12 GB + Virtual: 12 GB)
Storage	UFS 4.1 (256 GB variant)
Battery	6040 mAh (Global)
Process Technology	3 nm mobile SoC

The MediaTek Dimensity 9500 system-on-chip provides a heterogeneous big. LITTLE CPU architecture is suitable for parallel inference workloads. The availability of 12+12 GB RAM ensures that multiple quantized models (ASR, LLM, and TTS) can be loaded simultaneously without memory exhaustion. UFS 4.1 storage enables fast model loading and reduced initialization latency.

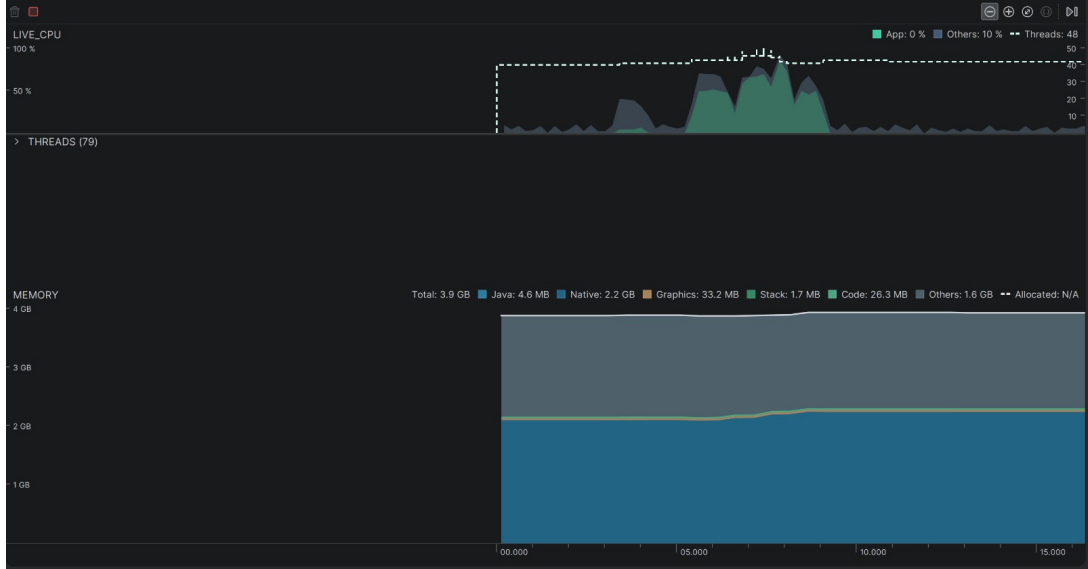
4.2 Measured Utilization and Latency

Resource usage was measured using Android Studio Profiler and system monitoring tools, while timestamped logs were used to compute end-to-end latency across the speech pipeline.

- **CPU Utilization and RAM Usage:** In Figure 2b, we can see that the CPU utilization is around 40% while running the Speech-to-Speech feature. At the same time, the memory usage is approximately 3.9 GB, as shown in the image. These values indicate that the app is using system resources at a normal level.



(a) Battery Usage



(b) Resource utilization measured using Android Studio Profiler during concurrent ASR, LLM, and TTS inference.

Figure 2: Performance evaluation showing battery consumption and resource utilization during concurrent ASR, LLM, and TTS inference.

- **Storage Footprint:** Quantized models substantially reduce on-disk size compared to full-precision checkpoints, enabling practical deployment within typical mobile storage constraints. The base application size is around 40 MB, and including the models, the total size is approximately 2.2 GB.
- **Battery Impact:** Short translation sessions do not cause significant thermal throttling. Power consumption increases during continuous inference but remains acceptable for real-world conversational usage. As shown in Figure 2a, the app consumed approximately 1400 mAh over about 3.5 hours.
- **Latency Metrics:** Timestamp analysis enables detailed measurement of each processing stage:

– *Speech Capture:* ~ 1.5 s (24,000 samples at 16 kHz; 225509.559–225510.702).

- *Whisper Transcription*: “How are you doing?” completed in ~ 2.1 s (225509.559–225511.766).
- *LLaMA Inference*: Typically < 1 s (e.g., token generation at 225512.779).
- *TTS Synthesis*: Dependent on text length; 951 ms generation time for 1099 ms audio (24,242 samples), 2126 ms for 1063 ms audio, and 2650 ms for 2298 ms audio.
- *TTS Playback*: Closely matches expected duration (e.g., 1084 ms actual vs. 1099 ms expected).

Overall, the hardware capabilities of the vivo X300 are sufficient to support fully offline, real-time speech-to-speech translation using quantized transformer-based models.

5 Optimization Techniques

5.1 Model-Level

Quantization All models are quantized:

- Whisper weights are INT8 quantised.
- Gemma-family weights are 4–8 bit.
- MMS-TTS can leverage quantized kernels in `sherpa-onnx` where available.

Quantization provides substantial improvements in latency and memory at small accuracy cost in conversational scenarios especially with Vivo X300 which has support for INT4 and INT8 .

Model size Whisper `tiny` checkpoints and smaller Gemma variants are chosen instead of larger configurations to keep per-utterance latency manageable.

5.2 System-Level Optimisations

Streaming Audio is processed in fixed 500 ms chunks with voice activity detection (VAD) triggering ASR on speech end. Llama translation tokens are streamed back to the UI via callbacks, enabling real-time partial translation display before sentence completion. This reduces perceived end-to-end latency from 9.4 s (full pipeline) to ~ 3 s (first tokens).

Threading Heavy computations are scheduled off the main thread:

- Model initialisation and disk I/O use `Dispatchers.IO` coroutines.
- `PipelineManager` runs ASR/translation/TTS synthesis on background threads.
- UI updates (transcription, live translation, status) are posted to `Dispatchers.Main` via callbacks, maintaining 60 fps responsiveness even during peak inference loads.

Caching Whisper and Llama models are loaded once at app startup via `nativeWhisperInit/ nativeLlamaInit` and held in memory across sessions. MMS-TTS models are cached per language in `MmsTtsManager.currentCode`; eviction occurs only on language change (e.g. `eng` \rightarrow `spa` \rightarrow `tam`), saving ~ 1.2 s reload time on subsequent utterances in the same language.

Audio Buffers

- **AudioRecord:** 16 kHz mono PCM, 8 K frame buffer (`getMinBufferSize`), 500 ms chunks (24 K–40 K samples per utterance).
- **AudioTrack:** 22.05 kHz mono PCM float, dynamic buffer `max(getMinBufferSize, 4×samples.length)` (20 K–77 K frames), preventing underruns while minimising playback delay (~ 1.4 s average).

6 Results and Discussion

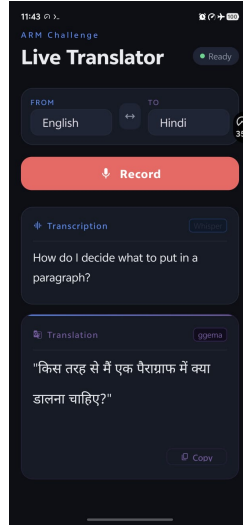


Figure 3: Mobile App

6.1 Latency and Quality

On the test device, end-to-end latency for medium-length utterances remains within a few seconds, split across ASR, LLM and TTS stages. Subjectively, Whisper provides sufficiently accurate transcriptions for everyday speech, while the Gemma-based translation is intelligible and MMS-TTS outputs natural-sounding audio for the languages provisioned.

6.2 Trade-offs and Limitations

Key trade-offs:

- Larger models improve accuracy but increase latency and RAM.
- Aggressive quantization reduces resources but can degrade quality for challenging inputs.

Limitations include:

- Language coverage restricted to MMS-TTS models present under `mms.tts`.
- No explicit personalization or domain adaptation.
- Robustness under heavy noise or strong accents depends on the chosen Whisper model.