AI-Driven Multilingual Voice Translator (HMI Project)

Raicu Bianca Elena, Rusen Andreea Emela,
Oteșanu Alexandra Maria

1. Introduction and Objectives

1.1 Project Scope

The scope of this project is to develop a high-performance, real-time voice translation application designed as a modern and adaptive interface. By leveraging the CustomTkinter framework, the system ensures a cutting-edge aesthetic and enhanced visual ergonomics. Our objective was to create an integrated system capable of capturing natural speech, transcribing it into text, and automatically translating it into a target language, followed by a high-quality neural audio synthesis of the translation.

The project emphasizes a seamless integration of these stages, allowing users to communicate rapidly without requiring technical knowledge of the underlying AI models. Through this application, we demonstrate how concepts from the Human-Machine Interface (HMI) laboratory can be applied to transform a standard computer into a sophisticated multilingual communication tool.

1.2 Addressed Problem

Our application addresses the communication barriers that arise when two individuals do not share a common language - a frequent challenge in travel, education, and international collaborations. Traditionally, manual text translation or dictionary usage significantly slows down and complicates a conversation.

This project tackles these barriers by automating the entire audio and text processing pipeline. The system eliminates the need for manual input, allowing users to speak freely while the software handles language detection, semantic translation, and synthetic voice generation. Consequently, the issue of linguistic isolation is solved through a continuous flow of audio-visual data, making human interaction simpler and more natural.

1.3 Project Objectives

- Transcription Accuracy: Integrating a state-of-the-art speech recognition engine (Faster-Whisper) capable of filtering background noise and accurately detecting the source language.
- Linguistic Flexibility: Implementing a translation system that supports at least five major international languages (Romanian, English, French, Italian, and Spanish).

- Real-Time Responsiveness: Optimizing the latency between the end of the recording and the audio playback by utilizing quantized models and asynchronous processing techniques.

- User Experience (UX/UI): Developing a modern graphical interface that supports theme customization (Dark/Light Mode) and provides dynamic visual feedback through custom widgets, ensuring a seamless interaction between the human user and the AI system.

- Data Persistence: Maintaining a session history to allow users to easily review and replay previous translations.

2. Team Roles and Task Distribution

The project was developed through close collaboration, adopting a modular development methodology. This approach allowed us to segment the application into clearly defined logical components, ensuring efficient technology integration and an equitable distribution of the workload. Each member contributed over 100 lines of code, covering specific development areas ranging from AI system architecture to signal processing and User Experience (UX) design.

Raicu Bianca Elena (AI Architecture & Neural Translation)

- **CustomTkinter Interface Configuration:** Implementation of the main application window and core system settings (Dark Mode, Dynamic Color Themes).
- **AI Model Architecture:** Developing the translation logic using Pivot Language strategies and optimizing neural model performance
- **NLP Pipeline Development:** Designing text processing functions and refining translation quality through Beam Search optimization techniques.

Rusen Andreea Emela (UI Design & Visual Control)

- **Layout and Organization:** Orchestrating visual elements using CTkFrame containers to create a fully responsive and adaptive interface.
- **Visual Feedback System:** Developing helper functions for real-time automated status updates and dynamic UI element transitions.
- **Theme Control & Navigation:** Implementing the Dark/Light Mode toggle and configuring specialized custom widgets (buttons, dropdown menus).

Oteșanu Alexandra Maria (Audio Processing & TTS Integration)

- **Audio Capture Management:** Configuring the sound capture stream at 16,000 Hz utilizing an asynchronous callback system.
- **TTS Integration (Speech Synthesis):** Implementing the audio playback module via Edge-TTS and managing temporary file overhead.
- **Data Persistence (Session History):** Developing the session storage system using Python Dataclasses and implementing the history-based Replay functionality.

3. System Architecture and Technologies

The application is built upon a sophisticated processing pipeline for audio signals and linguistic data, integrating three major technologies that collaborate to provide a seamless user experience.

3.1 Speech-to-Text (STT) with Faster-Whisper

The conversion of voice to text represents the first critical stage of the pipeline. It is powered by Faster-Whisper, a highly optimized implementation that enables speech recognition with minimal latency, even on hardware configurations lacking dedicated GPU acceleration.

This module integrates an automatic language detection feature that performs a spectral analysis of the initial seconds of the audio signal to identify the spoken language, eliminating the need for manual selection. To ensure high-precision transcription, we configured a Beam Search algorithm with a width of 5, allowing the model to evaluate multiple grammatical hypotheses in parallel and select the most coherent one.

Furthermore, the integration of a Voice Activity Detection (VAD) filter allows the system to ignore ambient noise or periods of silence. This prevents the generation of erroneous text fragments and optimizes computational resources by strictly processing active speech segments. To maintain low latency, the Faster-Whisper model is loaded onto a dedicated execution thread, ensuring that the CustomTkinter interface remains responsive and displays real-time status animations during processing.

3.2 Text-to-Speech (TTS) with Edge-TTS

Speech synthesis serves as the system's output module, responsible for converting the translated text into an audio response that naturally mimics human intonation. Edge-TTS technology utilizes advanced neural networks to generate fluid speech, far superior to traditional synthesis systems that often sound robotic or disjointed.

To achieve an authentic user experience, we implemented a linguistic mapping mechanism that associates each target language code with a specific vocal identity, ensuring correct accents and regional pronunciation. The entire generation process is orchestrated asynchronously using the asyncio library, allowing the application to communicate with neural synthesis servers without interrupting the GUI or blocking user interaction. The audio playback is triggered via a separate thread to prevent any "freezing" of the modern CustomTkinter (CTK) animations.

3.3 Signal Processing and Audio Management

The management of raw audio streams is the technical foundation of the application, ensuring professional-grade fidelity for both capture and playback. We standardized the sampling rate at 16,000 Hz (mono), which is the optimal format required by AI models to extract vocal features accurately without overloading system memory.

Recording is handled via a callback system that continuously collects small audio chunks into a temporary buffer, maintaining application responsiveness while the microphone is active. A crucial aspect of audio management is the safety mechanism that integrates the FFmpeg utility for dynamic format conversion. This tool automatically intervenes to repair or re-encode audio files generated by

the TTS engine in case of hardware incompatibilities, guaranteeing reliable playback across different systems. The audio capture is managed asynchronously, with the microphone state visually reflected through the dynamic color transition of the recording button (from primary blue to recording red).

4. Technical Implementation and Design

4.1 NLP Engineering and AI Models (Raicu Bianca Elena)

This section details the "brain" of the application, covering the orchestration of neural models and the linguistic processing pipeline.

1. Model Initialization and Optimization

The core technical foundation was the implementation of the load_models() function. To achieve an optimal balance between inference speed and resource consumption, I selected the Faster-Whisper model (configurable in accurate or fast modes). The implementation utilizes the compute_type="int8" parameter - a quantization technique that significantly reduces the RAM footprint without compromising transcription accuracy. For the translation engine, I integrated Transformer-based models from the MarianMT (Helsinki-NLP) family, which are renowned for their efficiency in Neural Machine Translation (NMT).

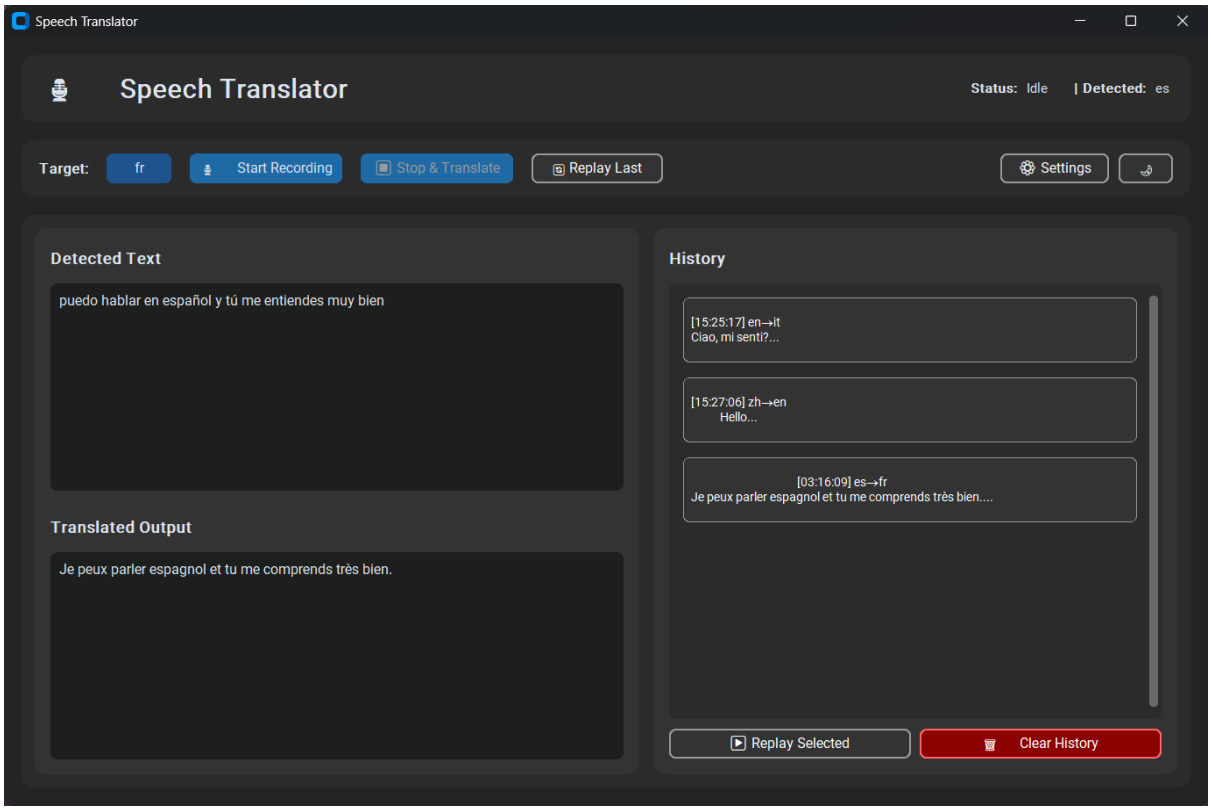
```python
def load_models(mode="accurate"):
    global tok_mul_en, mod_mul_en, tok_en_mul, mod_en_mul, whisper_model

    tok_mul_en = MarianTokenizer.from_pretrained(MODEL_MUL_EN)
    mod_mul_en = MarianMTModel.from_pretrained(MODEL_MUL_EN)
    tok_en_mul = MarianTokenizer.from_pretrained(MODEL_EN_MUL)
    mod_en_mul = MarianMTModel.from_pretrained(MODEL_EN_MUL)

    mod_mul_en.eval()
    mod_en_mul.eval()

    whisper_model_name = "tiny" if mode == "fast" else "small"
    whisper_model = WhisperModel(
        whisper_model_name,
        device="cpu",
        compute_type="int8"
    )
```

2. Implementation of the "Pivot Language" Strategy

A major challenge in this project was providing support for a wide range of languages without overloading system resources by loading dedicated models for every possible linguistic pair. The technical solution implemented was a cascading translation logic using English as an intermediary or "Pivot Language." If the system detects a source language other than English, the text is first processed through a multilingual model (mul-en) to bring it into a standardized form. Subsequently, this intermediate text is translated into the user-selected target language via the en-mul model. This architecture provides maximum flexibility, enabling complex translations between languages that lack a direct neural mapping (e.g., a fluid conversion from Spanish to French). Essentially, the pipeline normalizes the input to English before final synthesis, ensuring robustness across all supported pairs.

```
#Dacă limba engleză nu este limba sursă, traducem mai întâi în engleză
en_text = text if source_lang == "en" else _translate_mul_to_en(text, source_lang)

#Dacă limba engleză nu este limba țintă, traducem din engleză în limba țintă
if target_lang == "en":
    return en_text

return _translate_en_to_mul(en_text, target_lang)
```

In practice, the first stage of the pipeline 'normalizes' the input text by converting it into English (regardless of the source language), while the second stage determines whether to maintain the English output or route it through a subsequent translation model to reach the final target language.



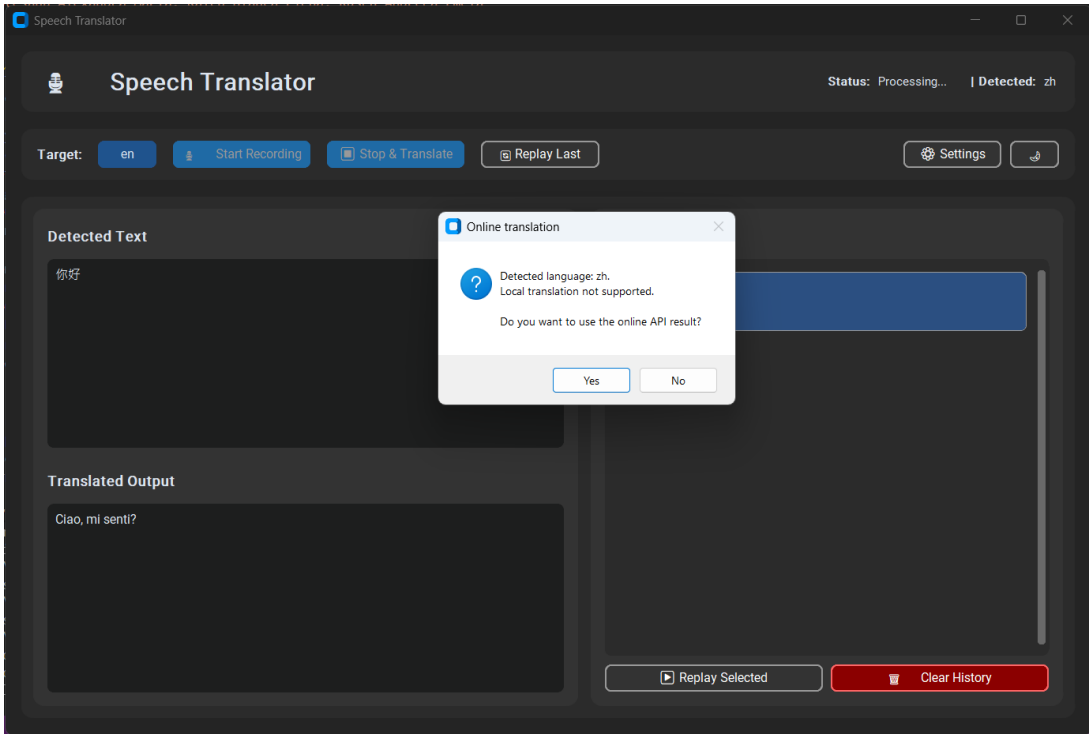
3. Translation Refinement through Generation Parameters

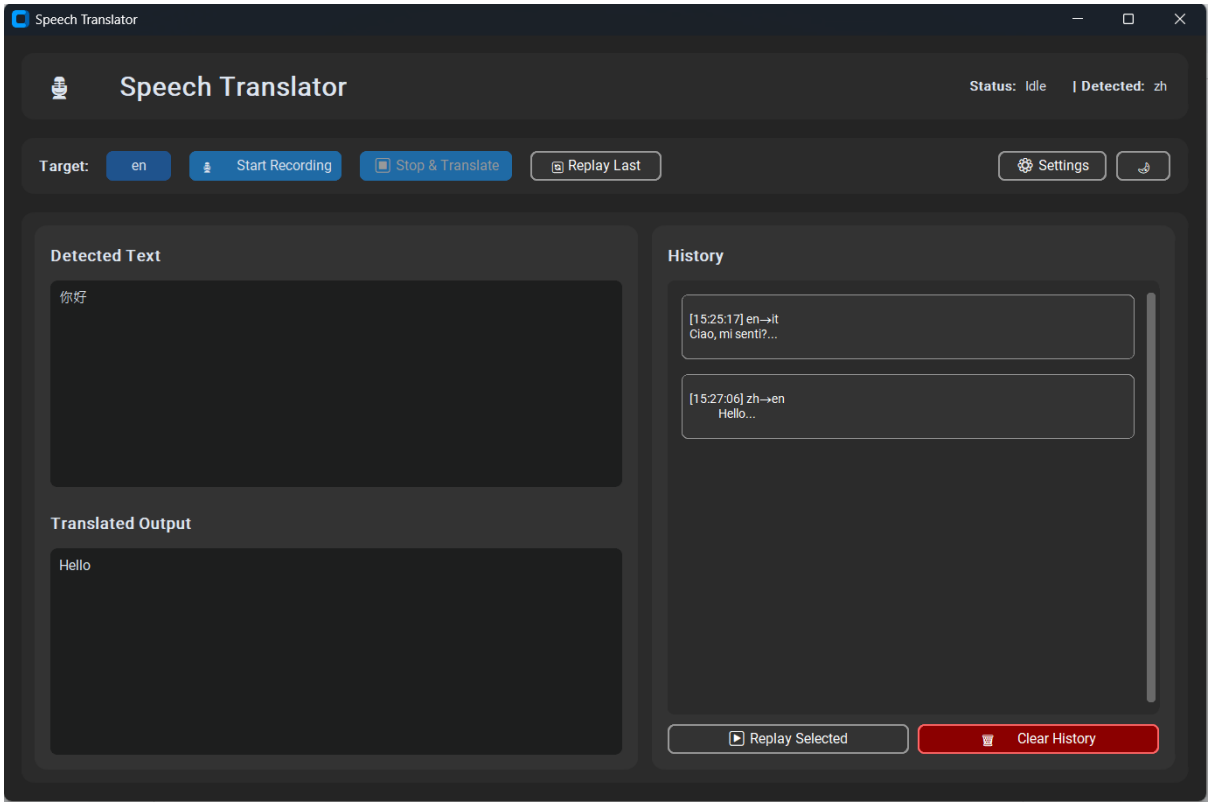
To ensure natural-sounding translations and mitigate common AI hallucinations, I configured the generation process using advanced optimization parameters. I implemented Beam Search with a width of 5, allowing the model to evaluate multiple sentence structures in parallel and select the most grammatically correct output. Additionally, I applied a repetition penalty (2.5) to prevent "looping errors" (the obsessive repetition of specific words), thereby ensuring the fluidity and coherence of the final output.

```python
def _translate_mul_to_en(text: str, detected_lang: str) -> str:
    inputs = tok_mul_en([text], return_tensors="pt", padding=True)
    out = mod_mul_en.generate(
        **inputs,
        max_length=256,
        num_beams=5,
        repetition_penalty=2.5,
        no_repeat_ngram_size=2,
        early_stopping=True
    )
    return tok_mul_en.decode(out[0], skip_special_tokens=True)
```

4. Exception Handling and Support for Extended Languages

To ensure system stability and versatility, the application has evolved from a simple filtering module into a hybrid processing architecture. Since the locally executed MarianMT models are optimized for a specific set of languages, I implemented a "Fallback API" logic. This mechanism automatically intervenes when the language detected by Whisper is not supported by the local models (e.g., Chinese, German, or Turkish).Instead of triggering an error or halting execution, the application intelligently redirects the request to an external translation engine (via Google Translate API). This technical solution ensures a seamless and continuous user experience, transforming local resource limitations into a global system capable of handling any linguistic pair, while simultaneously maintaining high speed and privacy for primary languages through local processing.






```
if used_fallback:
    # Întrebăm utilizatorul
    use_api = messagebox.askyesno(
        "Online translation",
        f"Detected language: {detected_lang}.\nLocal translation not supported.\n\n"
        "Do you want to use the online API result?"
    )
```

4.2 Signal Processing and TTS (Oteșanu Alexandra Maria)

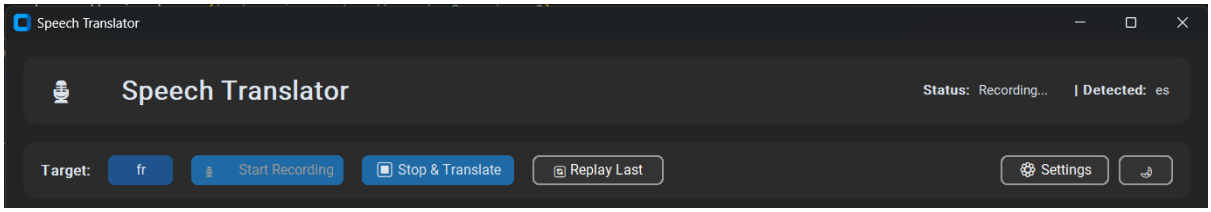
1. Signal Management and Callback Techniques

The technical foundation of the application is the audio capture system, configured at a sampling rate of 16,000 Hz (mono)—the industry standard for maximizing AI model accuracy. We implemented an asynchronous background callback function that collects audio fragments directly into a memory buffer (audio_buffer) only when the user activates the recording. This approach guarantees that the graphical interface remains fluid and responsive, eliminating the risk of window "freezing" during data acquisition.

```
def audio_callback(indata, frames, time, status):
    if recording:
        audio_buffer.append(indata.copy())
```

```
fs_local = 16000
```

```
stream = sd.InputStream(
    samplerate=fs,
    channels=1,
    callback=audio_callback
)
stream.start()
```



2. Speech Synthesis (TTS) and Temporary File Management

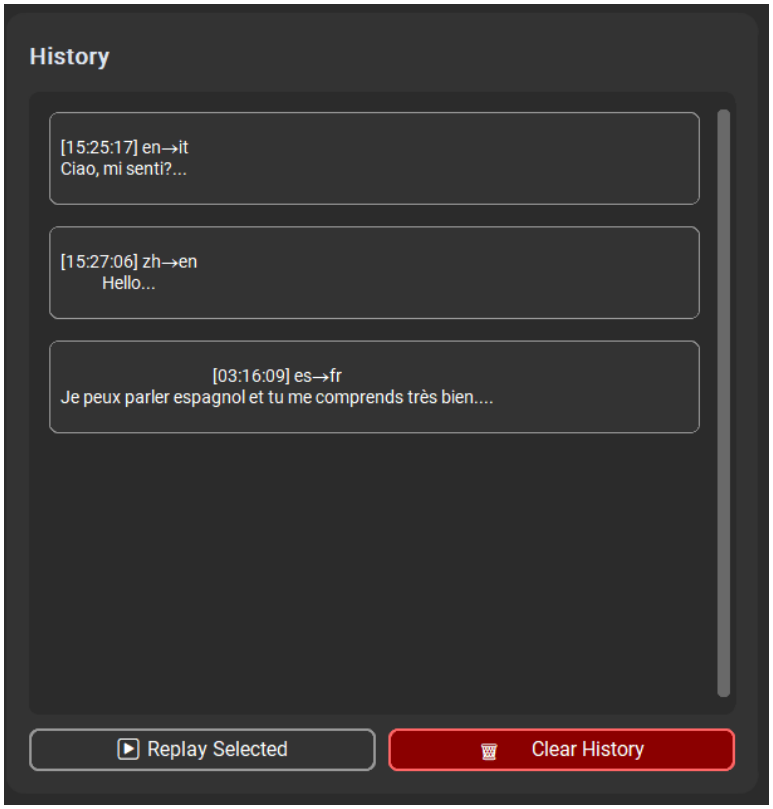
For the audio playback of translations, we integrated Edge-TTS technology, which leverages neural networks to generate fluid speech with natural intonation. We developed a file management mechanism that ensures minimal storage footprint: the application generates a temporary .wav file for each translation, plays it through the sound card, and uses a try...finally safety block to automatically delete the file immediately after playback. This "Cleanup" process prevents the accumulation of redundant data on the user's hardware.

```
def edge_speak_blocking(text, voice, rate=0, volume=0):
    tmp = tempfile.NamedTemporaryFile(delete=False, suffix='.wav')
    tmp_name = tmp.name
    tmp.close()
    fs_local = 16000
    try:
        try:
            asyncio.run(_edge_synthesize_to_wav(text, voice, tmp_name, rate, volume))
        except RuntimeError:
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)
            loop.run_until_complete(edge_tts.Communicate(text, voice).save(tmp_name))
            loop.close()

        with open(tmp_name, 'rb') as f:
            header = f.read(4)

        if header.startswith(b'RIFF'):
            if os.name == 'nt':
                winsound.PlaySound(tmp_name, winsound.SND_FILENAME)
            else:
                data, sr = sf.read(tmp_name)
                sd.play(data, sr)
                sd.wait()
        else:
            conv_tmp = tmp_name + '.conv.wav'
```

```
finally:
    try:
        if os.path.exists(conv_tmp):
            os.remove(conv_tmp)
    except Exception:
        pass
```

3. Data Persistence via the History System

To provide the user with full control over the working session, I implemented a HistoryItem data structure (utilizing Python dataclasses). This structure organizedly stores the original text, the translation, and the specific voice identity used. A key functionality is the Replay system, which allows for the re-listening of saved translations. To ensure an uninterrupted user experience, history playback is launched on a separate execution thread (threading), allowing the application to remain fully interactive while the audio is playing.




```python
@dataclass
class HistoryItem:
    ts: str
    src_lang: str
    tgt_lang: str
    original: str
    translated: str
    voice: str
```

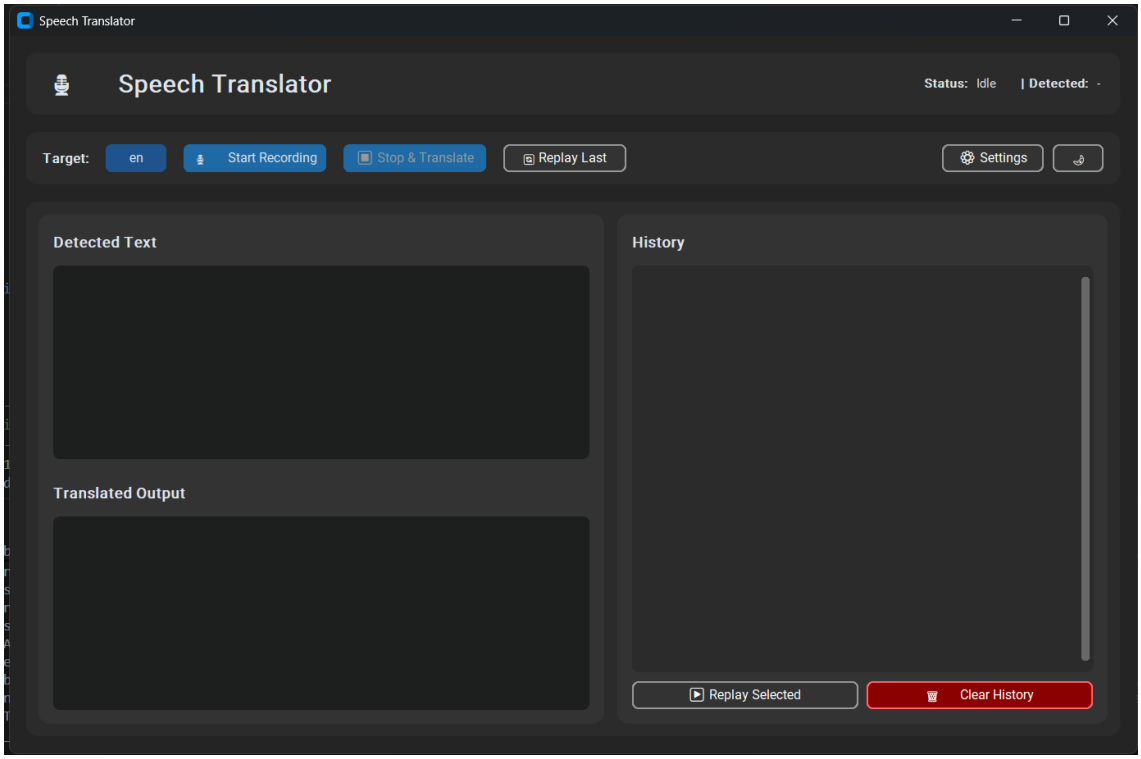
4.3 Visual Architecture and Control Logic (Rusen Andreea Emela)

1. Modular Layout and Frame-Based Organization (ttk.Frame)

We designed the interface using an Object-Oriented Programming (OOP) approach, structuring the application into specialized CTkFrame containers. This method allows for the logical separation of interaction zones:

- Sidebar: Manages theme settings and general system controls.
- Main Console: The central area featuring dedicated textboxes for source text and its translation.
- History Sidebar: A dedicated panel on the right side for viewing and replaying previous translations.

```python
def _build_main_content(self):
    """Zona principală cu text și istoric"""
    main = ctk.CTkFrame(self, corner_radius=12)
    main.grid(row=2, column=0, padx=16, pady=(8, 16), sticky="nsew")
    main.grid_columnconfigure(0, weight=2)
    main.grid_columnconfigure(1, weight=1)
    main.grid_rowconfigure(0, weight=1)

    # Left side - Text areas
    left = ctk.CTkFrame(main, corner_radius=12)
    left.grid(row=0, column=0, padx=(12, 6), pady=12, sticky="nsew")
    left.grid_columnconfigure(0, weight=1)
    left.grid_rowconfigure(1, weight=1)
    left.grid_rowconfigure(3, weight=1)
```



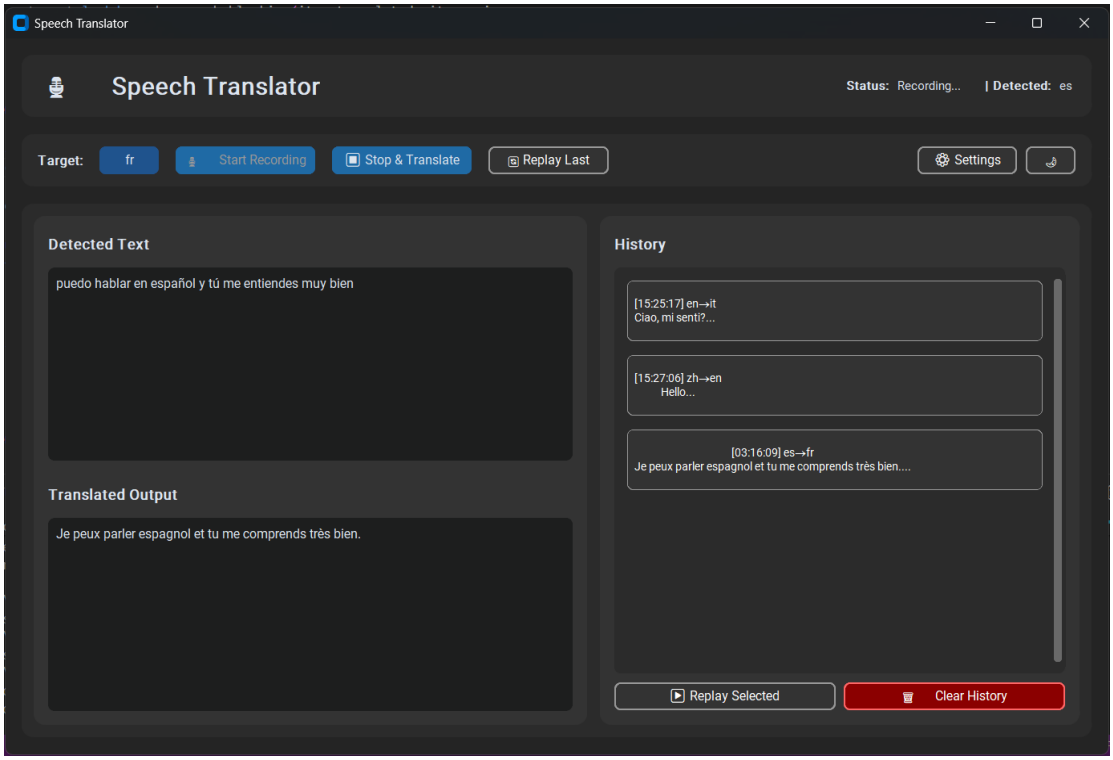
2. Helper Functions for UI Automation

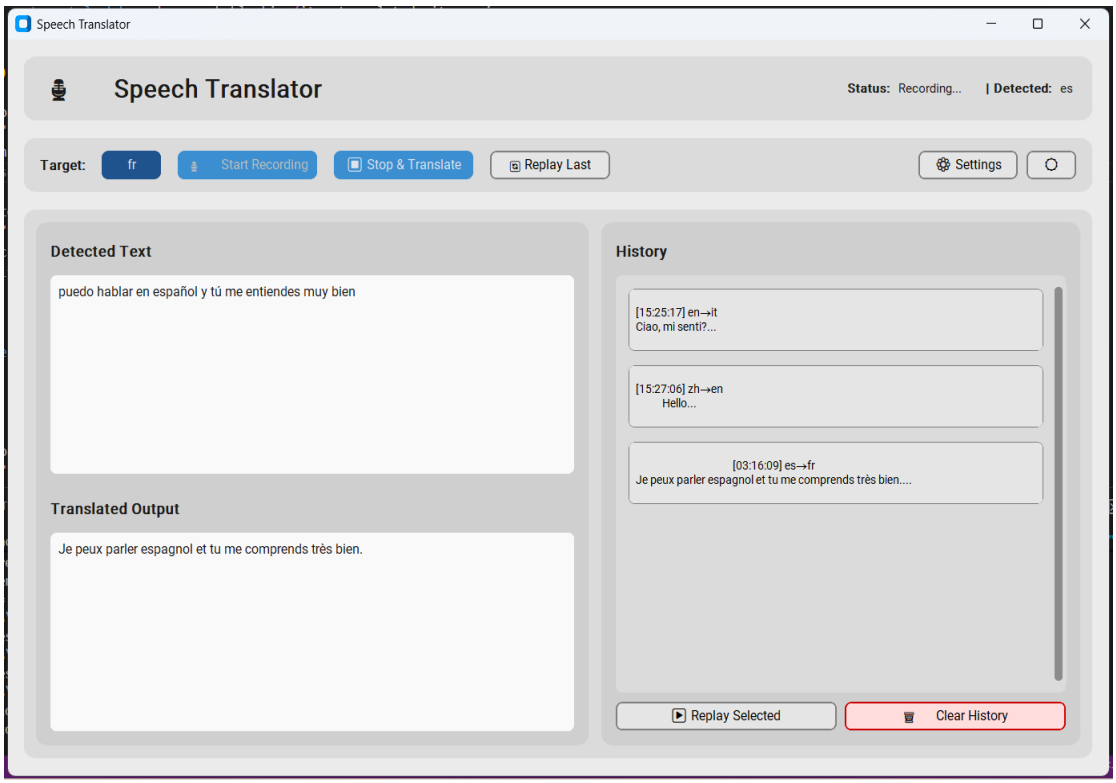
To maintain a clean codebase and avoid redundancy, we developed helper functions that centralize interface updates. These ensure that any change in the system state (e.g., transitioning from Recording to Processing) is instantaneously reflected in the UI by dynamically modifying labels, texts, and colors.

```python
def set_status(self, status: str):
    """Actualizează status-ul aplicației"""
    self.status_var.set(status)

def ui_set_text(self, widget: ctk.CTkTextbox, value: str):
    """Setează text în textbox"""
    widget.delete("1.0", "end")
    widget.insert("1.0", value)
```

3. Implementation of the Adaptive System (Dark/Light Mode)

We developed a toggle mechanism that allows for real-time switching of the visual theme. This function goes beyond changing the background color; it updates the styling of all integrated widgets (buttons, menus, textboxes), ensuring optimal legibility and a consistent aesthetic regardless of user preference.



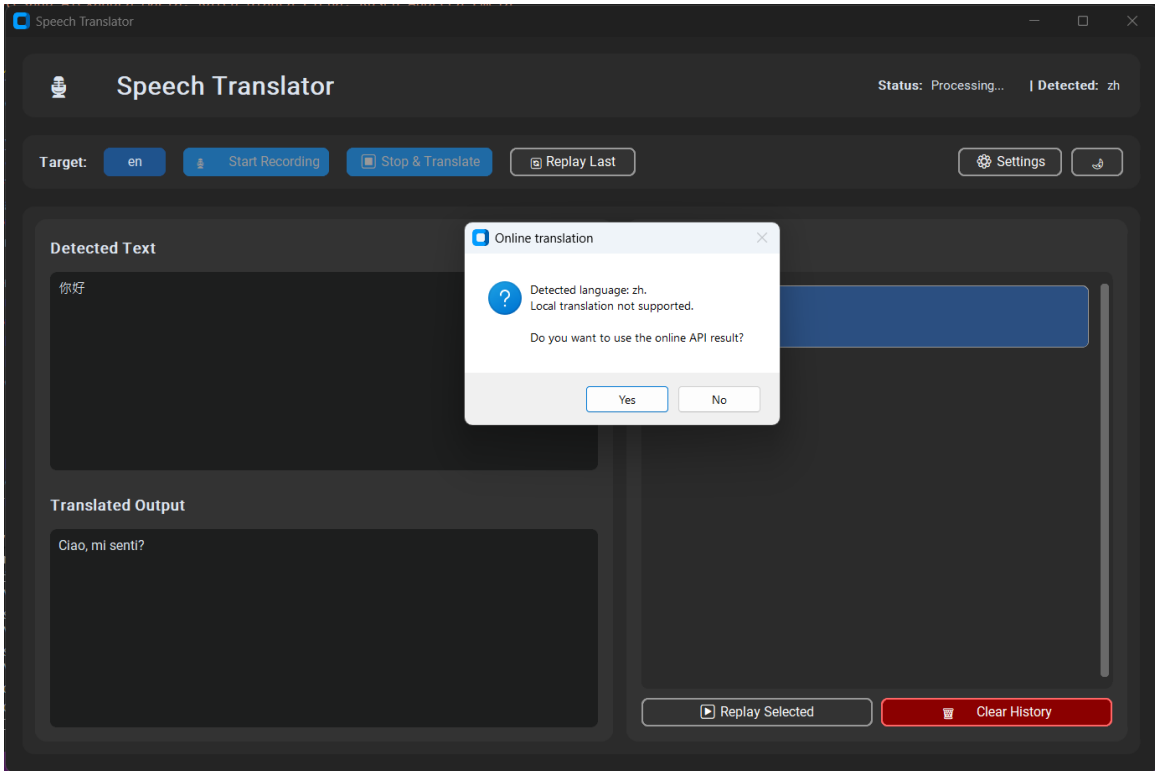


```python
def toggle_theme(self):
    """Schimbă între light și dark mode"""
    current = self.appearance_mode.get()
    if current == "dark":
        ctk.set_appearance_mode("light")
        self.appearance_mode.set("light")
        self.theme_btn.configure(text="🌞")
    else:
        ctk.set_appearance_mode("dark")
        self.appearance_mode.set("dark")
        self.theme_btn.configure(text="🌙")
```

5. Error Handling

5.1 Language Validation and Prevention of Erroneous Translations

The most critical error handled is the detection of a language that is not present in the local models. Instead of halting the application, I implemented a logical structure that redirects the text to an external API. This prevents a pipeline blockage and ensures the user receives a valid translation regardless of the linguistic context.




```python
if used_fallback:
    # Întrebăm utilizatorul
    use_api = messagebox.askyesno(
        "Online translation",
        f"Detected language: {detected_lang}.\nLocal translation not supported.\n\n"
        "Do you want to use the online API result?"
    )
```

5.2. Handling Resource Initialization Errors

Since audio processing and TTS playback are resource-intensive tasks, I implemented try...except blocks within the execution threads. For instance, in the edge_speak_blocking function, I handled the RuntimeError that occurs when an event loop is already active by creating a new loop to guarantee uninterrupted audio playback.

```
try:
    asyncio.run(_edge_synthesize_to_wav(text, voice, tmp_name, rate, volume))
except RuntimeError:
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    loop.run_until_complete(edge_tts.Communicate(text, voice).save(tmp_name))
    loop.close()
```

5.3. Data Flow Safety (Cleanup)

To avoid memory corruption or filling up storage space, we utilize a finally block at the end of each audio playback. This guarantees that the temporary .wav file is deleted even if the playback was interrupted by a system error or an abrupt shutdown of the application.

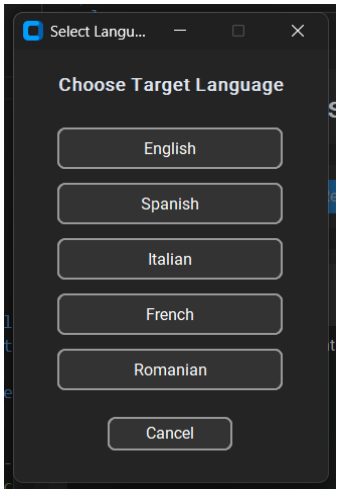
```
try:
    if os.path.exists(conv_tmp):
        os.remove(conv_tmp)
except Exception:
    pass
```

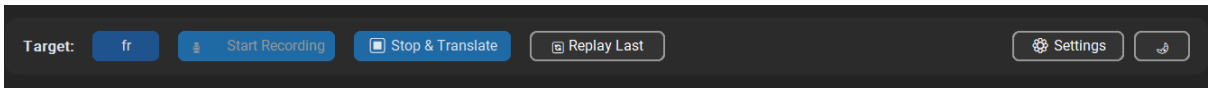
6. User Instructions

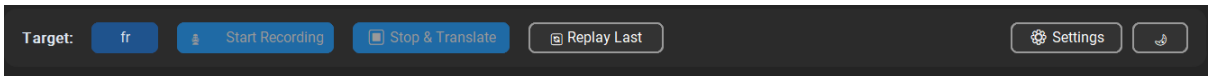
6.1. Primary Translation Workflow

To perform a translation, the user must follow four simple steps, each reflected by visual changes in the interface:

- **Language Selection:** Choose the target language from the dropdown menu (e.g., Italian).
- **Recording:** Press the **"Start Recording"** button. The button state changes, and the status bar displays *"Status: Recording..."*.
- **Processing:** Upon pressing the **"Stop + Translate"** button, all buttons become inactive (disabled) to prevent multiple conflicting commands, and the status updates to *"Status: Processing (AI)..."*.
- **Completion:** The transcribed and translated text appears in the dedicated textboxes, and the audio translation is automatically played.



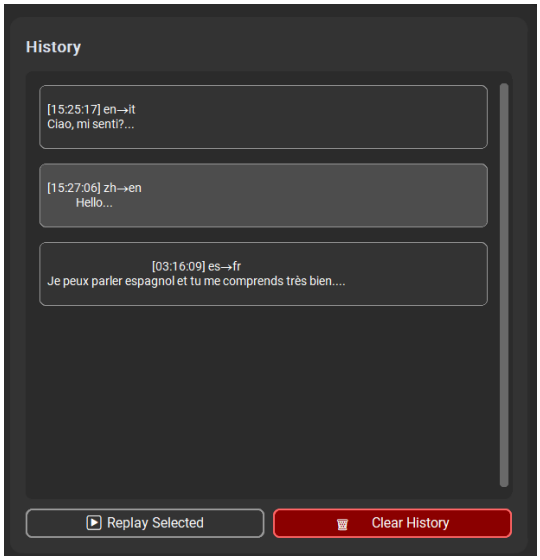




6.2. History Management and Audio Playback (Replay)

The application provides advanced session management features, located in the panel on the right side of the interface:

- Replay Last: Instantly replays the audio for the most recent translation.Selecție din Listă: Utilizatorul poate face click pe orice intrare din panoul din dreapta.
- List Selection: Users can click on any previous entry in the history panel.
- Replay Selected: After selecting an entry, this button triggers a dedicated execution thread that plays the saved audio without locking the user interface.
- Schimbarea Temei: The ◯/♫ button allows for instantaneous visual adaptation between Dark and Light modes.



7. Conclusions

The AI-Driven Multilingual Voice Translator successfully demonstrates the integration of complex neural models into a user-friendly, modern interface. By combining Faster-Whisper for robust speech recognition and MarianMT for cascading translations, we have created a system that balances local performance with global linguistic flexibility.

The project highlights the importance of Human-Machine Interface (HMI) principles, such as real-time visual feedback, asynchronous processing to maintain responsiveness, and inclusive design through adaptive themes. This development not only solves a real-world communication problem but also serves as a robust foundation for future implementations in embedded systems or mobile assistive technologies.

The source code available on GitHub has been updated with English documentation and comments for international accessibility, while the snippets presented in this document reflect the initial development phase.

8. Bibliography

- https://github.com/openai/whisper
- https://huggingface.co/Helsinki-NLP
- https://github.com/rany2/edge-tts
- https://github.com/SYSTRAN/faster-whisper
- https://python-sounddevice.readthedocs.io/
- https://docs.python.org/3/library/tkinter.html
- https://huggingface.co/docs/transformers/index
- https://ffmpeg.org/documentation.html
- https://customtkinter.tomschimansky.com/
- https://customtkinter.tomschimansky.com/documentation/