Prototype description

DSAI08

Noam Favier	(i6349778)
Daan Vankan	(i6331424)
Alexandros Ntoouz/Dawes	(i6323296)
Anne Katarina Zambare	(i6365696)
Paul Elfering	(i6365814)
Vladislav Snytko	(i6363101)
Evi Levels	(i6368803)

Maastricht - October 2025
KEN3300 - Project 3-1

Overview

The prototype implements a multimodal processing pipeline designed to analyze student communication and collaboration while working with EDMO robots. It integrates multiple microservices that are each responsible for a specific aspect of the analysis. They are coordinated by a central Go-based controller. Audio recordings are automatically transcribed, diarized, and segmented into conversational windows. Within each window, the system extracts verbal (NLP and emotion analysis) and nonverbal (speech timing and overlap) features, as well as robot movement data. These features are then combined, clustered, and visualized to uncover patterns in students' communication strategies, emotional expressions, and interaction dynamics.

What we have learned

Through developing the prototype, we have gained valuable insights into the multimodal analysis of student communication that will guide us in the next phase of the project. We learned how verbal content, emotional tone, nonverbal speech patterns, and robot movement can be captured, quantified, and integrated to uncover communication strategies. This experience gives us confidence in our ability to design and implement an effective system in the upcoming project phase.

How to run

To run the system locally, the microservices are started through Docker Compose, while the Go pipeline itself is executed on the host. First, start all services in the background and build them if needed:

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

Once the services are running, the pipeline can be executed by providing an audio file and selecting the configuration environment (e.g., `dev`):

```
CONFIG_ENV=dev go run ./src/go_core <path/to/wav>
```

Here, `CONFIG_ENV=dev` instructs the pipeline to load the development configuration file from `config/dev/config.yaml`, and `./docs/jfk.wav` is the input audio file to be processed. Any valid `.wav`, `.mp3`, or `.m4a` file can be used in place of the example.

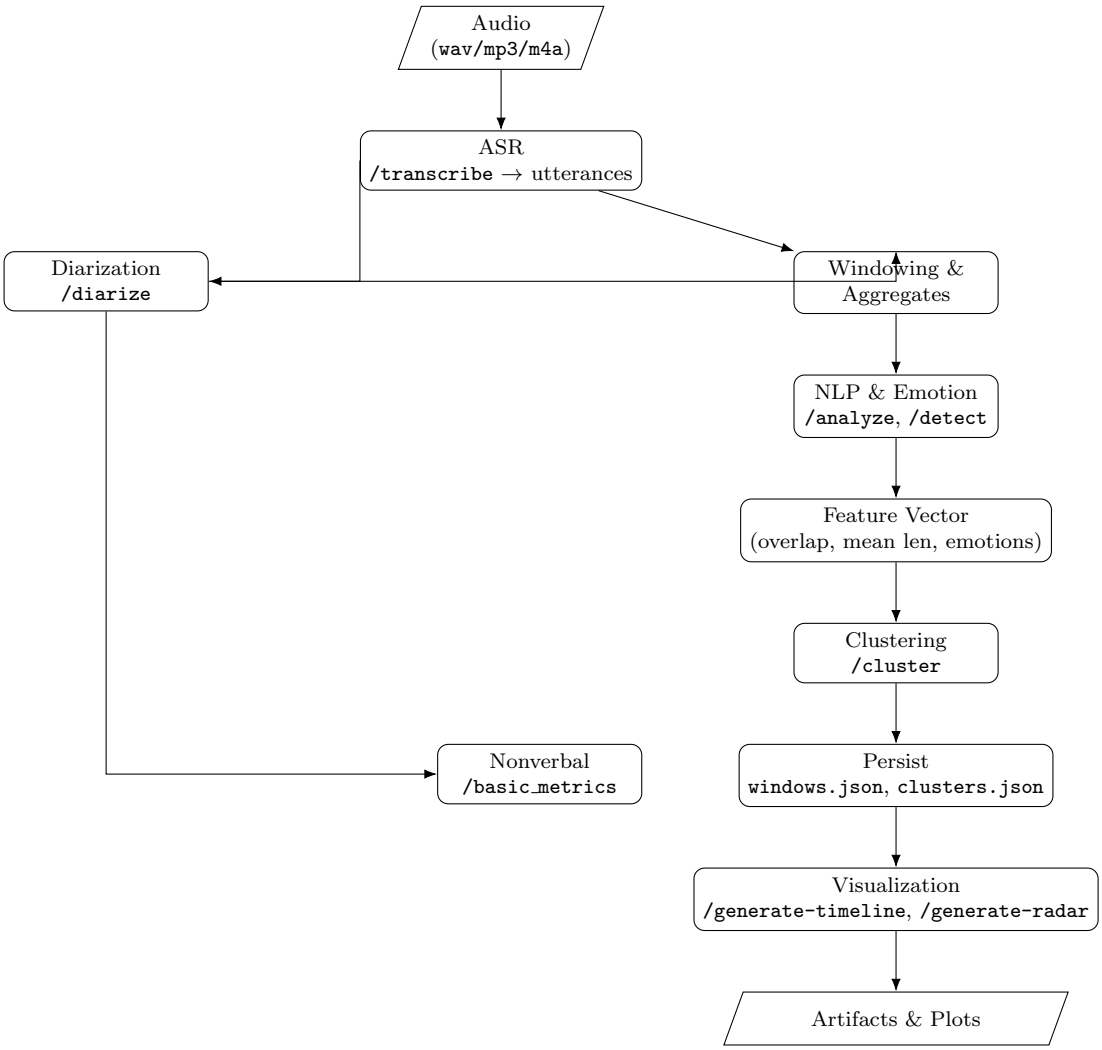
Example:

```
CONFIG_ENV=dev go run ./src/go_core ./docs/jfk.wav
```

Pipeline

At a high level, the Go-based `edmo-pipeline` is a coordinator: it reads settings from YAML (`config/dev|prod/config.yaml`), then `main.go` builds a `Pipeline` and runs it on the given audio file. A single, long-timeout HTTP client (`clients.NewHTTP()`) talks to a set of microservices. First, we transcribe the audio via ASR (`/transcribe`) to get timestamped lines of text. If a diarization service is configured, we also call `/diarize` and attach speaker IDs to those lines. Next, we slide an overlapping time window across the conversation, compute simple aggregates (who talks how much, how much overlap), and per window call NLP (`/analyze`) and Emotion (`/detect`) to score the content. Each window is then turned into a numeric feature vector (e.g., overlap rate, mean utterance length, a few emotion scores), and the full matrix goes to the clustering service (`/cluster`), which may also reduce dimensionality. We persist everything to a timestamped output folder (`windows.json`, `clusters.json`), and ask the visualization service to produce a timeline and a radar chart (`/generate-timeline`, `/generate-radar`). All HTTP responses are decoded into typed Go structs, errors include the server's message body for easier debugging, and any service can be disabled by leaving its URL empty.

- **Config**: `config.Root` (`pipeline`, `audio`, `services`, `features`, `paths`); dev/prod variants; helpers like `DurSeconds`.
- **Transport**: `clients.HTTP` wraps `http.Client` (timeout: 30 min) and provides typed methods: `ASR`, `Diarize`, `NLP`, `Emotion`, `Cluster`, `GenerateTimeline`, `GenerateRadar`, `NonverbBasicMetrics`.
- **ASR**: multipart upload of the original audio; returns ordered segments {`start,end,text`} and language code.
- **Diarization**: optional; returns {`start,end,speaker`}; aligned to utterances via longest-overlap assignment.
- **Windowing**: sliding windows (`features.time_window`, `features.overlap`); compute speaking-share, overlap-rate; aggregate emotion scores per window; build feature vector.
- **Nonverbal metrics**: if diarization present, compute conversation- and speaker-level metrics (`/basic_metrics`) incl. overlap/silence ratios & interruptions.
- **Clustering**: send feature matrix; receive labels, membership matrix, reduced components; log shape and explained variance.
- **Persistence & Viz**: write JSON artifacts to a session directory; request timeline/radar plots to the viz service with output paths.



Automatic Speech Recognition

The ASR microservice implements a lightweight ASR service using `FastAPI` together with OpenAI's `Whisper` model. The model loads once at startup, relying on the `WHSIPER_MODEL` environment var to chose the desired architecture (default: `base`). The transcription endpoint receives an uploaded file, temporarily stores it using a secure OS-level temporary file, and processes it with Whisper. The system optionally enforces a fixed language via `WHISPER_LANG`; otherwise, Whisper performs automatic language detection. The output is then structured into Pydantics models to ensure schema consistency and OpenAPI doc.

- **Input:** Audio file via `POST /transcribe` (`.wav`, `.mp3`, `.m4a`).

- **Preprocessing:** Uploaded file written to a temporary file and validated for supported formats.
- **Inference:** `model.transcribe()` executed with `fp16 = false` for CPU compatibility.
- **Output:** A list of timestamped segments (`start`, `end`, `text`) and detected language.

The temporary file is always removed in a `finally` block to avoid data leak. Furthermore, keeping the model as a global singleton prevents repeated some initialization overhead, making the service efficient most jobs, including batch and large files.

NLP features extraction

The NLP microservice is capable of extracting several Natural Language Processing (NLP) features to analyze what the students are actually saying. Specifically, the service can:

- **Preprocess text**: Perform basic cleaning tasks such as converting to lowercase, removing punctuation, and normalizing spaces to prepare the text for further analysis.
- **Extract keywords and key phrases**: Identify the most relevant words or phrases from the students' speech, helping highlight important topics.
- **Perform sentiment analysis**: Assign a sentiment label to each piece of text ("Very Negative", "Negative", "Neutral", "Positive", or "Very Positive") and thereby providing insight into the sentiment of the students' communication.
- **Generate sentence embeddings**: Convert text into high-dimensional vector representations using a sentence transformer model, enabling tasks such as similarity analysis and clustering.

Emotion analysis

The emotion microservice complements the NLP analysis by identifying the emotional tone of speech content. It uses a fine-tuned transformer model to:

- **Detect multiple emotions**: Identify a range of possible emotions, specifically anger, disgust, fear, joy, neutral, sadness, and surprise.
- **Assign emotion confidence scores**: Quantify how strongly each emotion is expressed.
- **Determine the dominant emotion**: Identify the most likely (dominant) emotion based on the highest confidence score.

Nonverbal Features Extraction

As part of the overall processing pipeline, we extract a range of nonverbal speech features for each student, as well as metrics that describe the conversation as a whole. These features will later be used in data analyses aimed at uncovering communication strategies employed by the students. Nonverbal cues such as speaking duration, pauses, and interruptions provide valuable insights into interaction dynamics and collaboration patterns that are not captured by verbal content alone.

At this stage, the set of extracted metrics includes only basic features, implemented primarily to test the functionality of the pipeline. More advanced features related to pitch, prosody, and energy will be incorporated at the beginning of Phase 2. The current list includes:

- **Speaker-level features (per speaker):**
  - **total_speaking_duration**: Total time (in seconds) the speaker was speaking during the conversation.
  - **total_turns**: Number of speaking turns taken by the speaker.
  - **speech_ratio**: Proportion of conversation time occupied by the speaker's speech: total_speaking_duration/conversation length.
  - **mean_turn_duration**: Average duration of the speaker's turns.
  - **median_turn_duration**: Median duration of the speaker's turns.
  - **std_turn_duration**: Standard deviation of turn durations.
  - **min_turn_duration**: Shortest turn duration.
  - **max_turn_duration**: Longest turn duration.
  - **percentiles**: Dictionary of turn duration percentiles (e.g., 25th, 50th, 75th).
  - **interruptions_made**: Number of times the speaker interrupted another.
  - **interruptions_received**: Number of times the speaker was interrupted.
  - **interrupted_by**: Mapping of other speakers to the number of interruptions they made toward this speaker.
- **Conversation-level features (single per dialogue):**
  - **num_speakers**: Total number of distinct speakers in the conversation.
  - **total_speaking_time**: Sum of all speakers' speaking durations (including overlaps).
  - **overlap_duration**: Total time segments where two or more speakers talked simultaneously.

- **silence_duration**: Total time with no speech activity.
- **overlap_ratio**: Proportion of conversation time containing overlapping speech: overlap_duration/audio length.
- **silence_ratio**: Proportion of conversation time that was silent: silence_duration/audio length.
- **total_interruptions**: Total number of interruptions across all speakers.
- **interruption_rate**: Average number of interruptions per minute of conversation.

Robot Data Logs Analysis

The pipeline begins with `raw .log` files capturing:

- Timestamps of control switches
- User actions
- Oscillator changes

The robot data microservice processes these logs by importing and scanning the data, aligning timestamps, and converting it into structured timelines (`timeline.csv`). It then extracts features such as correlation matrices, Principal Component Analysis (PCA), and K-Means clustering to reveal behavioral patterns and control dynamics between students.

The service also generates visual summaries including control timelines, event frequency charts, and temporal evaluation plots. Shared utility functions provide common logic used throughout the processing steps.

Robot Movement Tracking

The movement tracker microservice is responsible for extracting quantative movement data from the EDMO robots recorded during the classroom sessions. It processes raw `.mp4` video files containing overhead recordings of the robot equiped with ArUco markers and generates time-based features that describe the robot's trajectory and dynamics.

The script performs the following main steps:

- **Video import and preprocessing**: Loads the raw video using OpenCV and converts each frame to grayscale to optimize marker detection.
- **Marker detection**: Detects ArUco markers and identifies their unique IDs to identify the robot's position. For each frame, the algorithm extracts the center coordinates (x, y) of the detected marker.

- **Position tracking and visualization**: Records the robot's position over time and generates a visual trail showing its recent movement to support qualitative inspection of tracking accuracy.
- **Velocity computation**: Calculates the frame-to-frame velocity components (v_x, v_y) and total speed $v = \sqrt{v_x^2 + v_y^2}$ in both pixels per second and centimeters per second, based on a user-defined pixel-to-centimeter ratio.
- **Data export**: Stores the processed motion data as a structured `.csv` file containing timestamps, positions, and velocity components for further integration with communication and behavioral analysis pipelines.

Dimensionality reduction and Clustering

The clustering microservice has two purposes: it performs dimensionality reduction on the available dimensions and then performs clustering the on the datapoints in the resulting space.
For the dimensionality reduction the service supports the choice between two options:

- **Principal Component Analysis**: This dimensionality reduction method conserves as much variance as possible within a given number of dimensions. The dimensions are orthogonal to each other and therefore do not share information.
- **Sparse Principal Component Analysis**: This dimensionality reduction reduces into dimensions that are simpler to interpret, but are not orthogonal to each other. Therefore the resulting dimensions can incorporate parts of the same information.

The explained variance per reduced dimension is then displayed in a graph, to assess which amount of dimensions provides effective results.

For the clustering we implemented **fuzzy c means clustering**, a clustering algorithm that allows points to belong percentual to several clusters, therefore generating a confidence that the point is in a given cluster. Visualization of the actual clustering results is not implemented in the prototype, but the radar chart provides a insight into how an interpretation of these results may look.