Whisper Transcription Pipeline

1. Project Overview

Objective: Build an automated audio-to-text transcription system using OpenAI's Whisper model. The system must accept recorded audio files, convert them if necessary, transcribe the speech using Whisper, and output structured transcripts (TXT/JSON).

The design supports audio files up to 60 minutes and produces high-accuracy transcripts. Key innovations have been introduced to improve robustness, contextual understanding, and output usability.

2. Input Specs and Assumptions

These define the input conditions, such as file format, language support, and duration limits. Smart chunking and LLM summarization are enabled post-transcription.

Category	Original Assumption	Suggested Additions / Improvements
Input Format	Audio/video formats via ffmpeg	'.flac', '.wav', '.mp3', '.m4a', '.aac', '.ogg', '.webm', '.opus', '.mp4', '.mov', '.mkv', '.avi' -> Normalized using `ffmpeg to 16kHz, mono, 16-bit PCM WAV (Whisper-compatible)
Language	English (extendable)	Auto-detected using Whisper; supports over 90 languages. Summarization assumes English-only output (current models are English-centric)
Max File Duration	Up to 60 minutes	Longer files will be rejected or chunked with warnings
Max File Size	≤100MB	Preprocessing ensures optimal memory usage and performance; larger files tested on GPU.
Speaker Overlap	Currently not handled	Diarization (`--speaker`) available for future integration using WhisperX + pyannote.audio
Chunk Strategy	Smart Chunking via silence detection	Uses pydub for silence detection and contextual padding. Fallback to fixed-length chunking if silence not detected.
Summary Mode	Post-transcription using LLMs such as Mistral	Optional summarization via --summary flag. Default: Mistral-7B-Instruct. Custom Hugging Face models accepted via --summarized-model

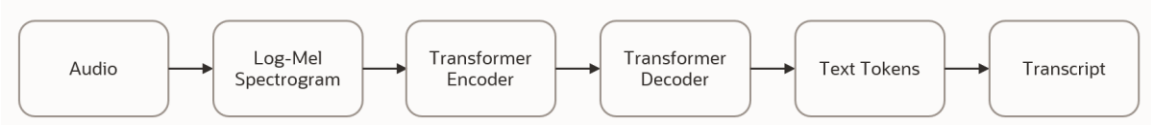
	Output Format	.txt, .json	.json includes transcript, summary, metadata (e.g. language, model used, chunk map). Additional .log or .pdf formats available for reporting
	Upload Method	CLI/File drop to Object Storage	REST API endpoint /transcribe returns structured JSON output. CLI/batch processing remains supported

3. Whisper Model Evaluation Plan

3.1 Whisper Models

Whisper's model family allows users to select models based on compute availability, latency needs, and transcription quality targets. This flexible design enables both lightweight and high-accuracy deployment options, from edge devices to GPU-accelerated backends.

Whisper is a family of encoder-decoder transformer models trained for automatic speech recognition (ASR) on multilingual, multitask data. All Whisper models share the same overall structure, but differ in size and performance. Below is simplified diagram from the Whisper model:



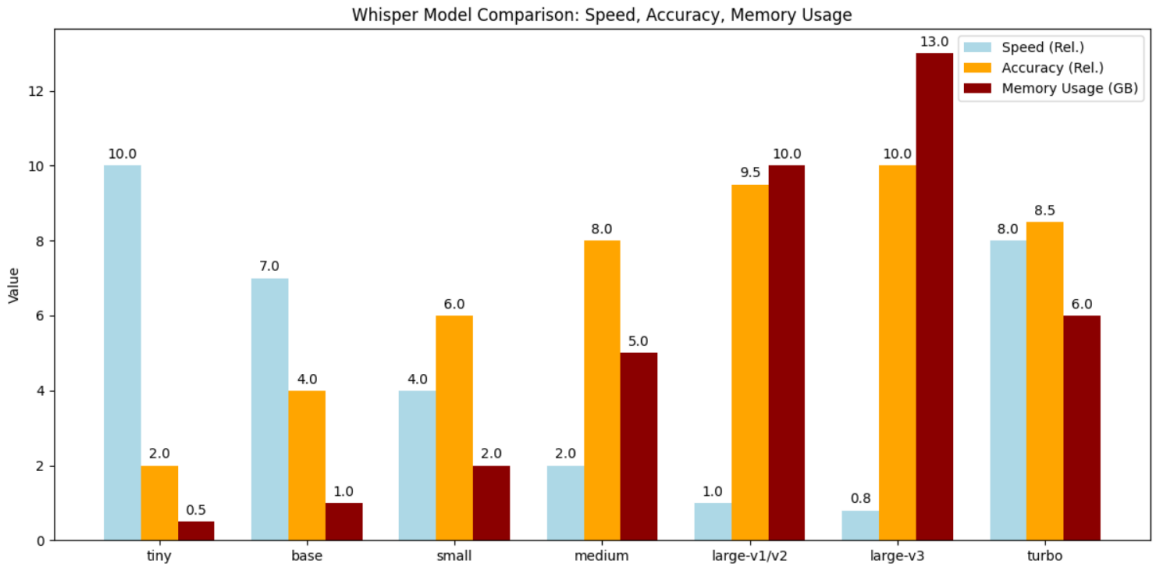
3.2 Whisper Models Evaluations

Each model is evaluated based on speed, accuracy, and GPU/memory requirements. This helps determine suitability for different workloads or environments.I evaluated the following Whisper models:

Model	Parameters	Architecture	Speed	Accuracy	Use Case	GPU Requirement	Memory Usage (est.)	Max Audio Length
tiny	39M	6L encoder, 6L decoder	Fastest	Lowest	Quick tests, edge inference	CPU/GPU	~0.5 GB	~10–20 mins
base	74M	6L encoder, 6L decoder	Very fast	Moderate	Real-time transcription	CPU/GPU	~1 GB	~30 mins
small	244M	12L encoder, 12L decoder	Fast	Good	Podcasts, voice notes	CPU/GPU	~2 GB	~60 mins
medium	769M	24L encoder, 24L decoder	Moderate	High	Meetings, analysis	GPU preferred	~4–6 GB	~90 mins
large-	1.55B	32L	Slow	Best	Multilingual	GPU	~10GB	~2 hours

v1/v2		encoder, 32L decoder			, production	(16GB+)		
large-v3	1.55B	32L encoder, 32L decoder	Slower (GPU only)	Highest	Noisy, long-form, multilingual	GPU (A100+)	~12-14 GB	~2 hours+
turbo	809 M	Optimized version of Large-V3	Ultra fast (optimized for H100)	Good	Low-latency use	H100 or A100	~6GB	60+ mins (chunked)

Below is a simple graph of my evaluations:

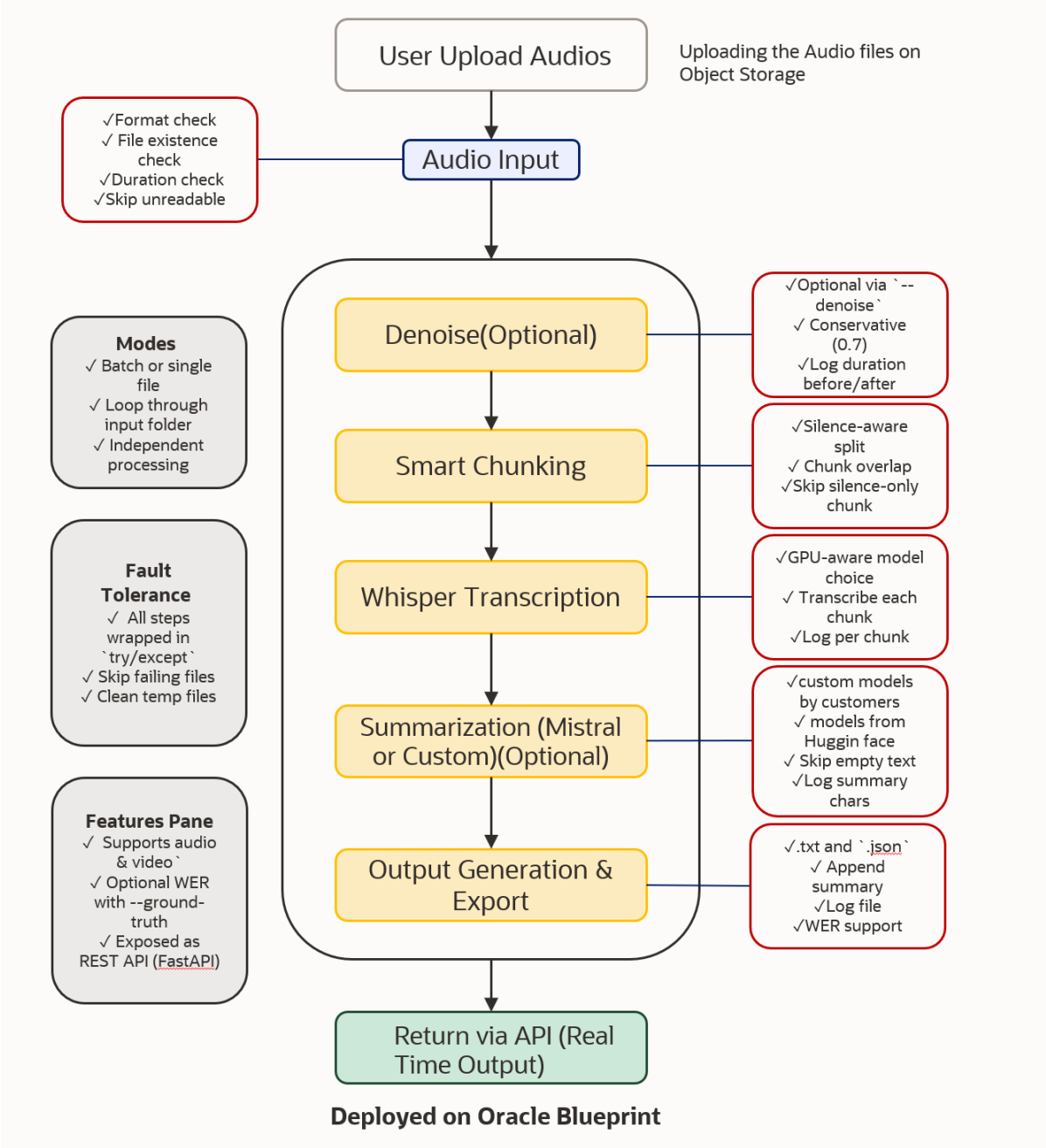


3.3 Whisper Models Recommendations

- All models are trained end-to-end using the same loss function and multitask objectives (transcribe, translate, language ID, timestamp prediction).
- Instead of CTC, Whisper uses standard cross-entropy loss over the output token sequence. Each sample is prefixed with a task-specific token such as `<|transcribe|>` or `<|translate|>` to condition the model accordingly.
- Larger models consistently perform better across noisy, accented, and multilingual data.
- `tiny` and `base` models are useful for mobile or edge inference.
- `small` and `medium` provide a good balance for production use.
- `large-v3` offers the best performance but requires substantial memory (≥10GB) and is slower to transcribe.
- `Turbo` model is an optimized version of large-v3 that offers faster transcription speed with a minimal degradation in accuracy

4. Architecture Diagram

The diagram below shows how audio moves through various stages: chunking, transcription, summarization, output and will return via API. Each of these steps will be explained as follows:



4.1 Audio Files input (User Upload Audios and Audio Input)

To ensure Whisper can handle a wide range of real-world audio and video sources, we designed a robust, flexible ingestion pipeline. The system supports both audio and video files, standardizes them to a format compatible with Whisper, and performs validation to ensure smooth downstream processing. In details:

- Extended SUPPORTED_EXTENSIONS to include audio (.flac, .mp3, .wav, .m4a, .aac, .ogg, .opus, .webm) and video (.mp4, .mov, .mkv, .avi).
- Rewrote `convert_to_wav()` to use `ffmpeg`, which can extract and normalize audio from both audio and video files.
- Standardized all inputs to 16kHz, mono, 16-bit PCM WAV — the format expected by Whisper.

We selected `ffmpeg` due to its broad format support(supporting nearly every known audio/video codec), performance, and reliability. ffmpeg is an industry-standard tool offers precise control over sample rate, channels, and bit depth. This design maximizes flexibility, allowing customers to transcribe speech from almost any source: local audio, and video recordings. Below is a table comparing all our options:

Method	Pros	Cons
ffmpeg subprocess	Universal, works for all formats, fast	Requires ffmpeg to be installed
AudioSegment (pydub)	Simple Python API, okay for .mp3/.wav	Fails with video, needs ffmpeg anyway, slower
moviepy	Handles both audio/video	Heavy, slower, uses ffmpeg under the hood
sox	CLI audio processing	Limited video support, extra dependency

All input files go through a series of quality gates to improve reliability and ensure Whisper receives valid input:

- **Format check:** ensure only supported audio/video types are processed
- **File existence check:** skip missing or zero-byte files
- **Duration check:** reject files that exceed the max runtime (default: 60 minutes)
- **Unreadable file handling:** corrupt or malformed files are skipped with warning logs

By combining smart input filtering with robust format normalization, this component ensures that transcription jobs remain resilient, scalable, and user-friendly — even in noisy or unstructured environments.

4.2 Denoising (Optional)

In real-world scenarios, audio is rarely clean. Whether recorded in meeting rooms, interviews, or social gatherings, background noise significantly impacts transcription accuracy. To address this, we implemented a high-fidelity denoising strategy that enhances speech clarity before it reaches the Whisper model. This preprocessing step improves signal-to-noise ratio, reduces hallucinations, and leads to more accurate transcripts and summaries.

Our approach is based on a two-stage denoising pipeline. First, Demucs is used to isolate structured background interference like music or ambient sound. It performs especially well in acoustically busy environments. Next, noisereduce targets non-stationary static noise such as fan hum or hiss. The combination handles both structured and broadband noise effectively. This structure is optimal because it is modular and fault-tolerant. If Demucs fails or crashes, the original input is used as a fallback. It is also well-tuned for real-world applications like Zoom calls, YouTube interviews, and podcasts. Users can adjust `prop_decrease` to control noise suppression aggressiveness, and detailed logging ensures traceability. The process is optional and only triggered via the `--denoise` flag, providing flexibility for customers who want cleaner input without enforcing it for all workloads.

We chose Demucs and noisereduce because together they provide a balanced, extensible solution tailored for transcription pipelines. Demucs excels at vocal isolation, while noisereduce offers lightweight spectral gating. Compared to alternatives, RNNoise is optimized for real-time static noise but struggles with dynamic input and lacks Python-native tooling. Spleeter is good for music editing but introduces artifacts in noisy speech. DeepFilterNet offers high-quality filtering but comes with C++/ONNX complexity. Voicefixer is strong at restoration but less suited for general-purpose denoising. Our chosen stack delivers accuracy, maintainability, and adaptability for noisy real-world inputs.

Tool	Pros	Cons
Demucs	• Excellent for music separation • Isolates vocals cleanly • Works well in noisy, complex environments	• Slower than others • Requires GPU for best results
noisereduce	• Lightweight • Easy to use with librosa • Good for broadband/static noise	• Assumes noise is non-speech • Fails in party/bar scenes
Spleeter	• Fast and GPU-accelerated • Good for music editing	• Not tuned for noisy speech • Less accurate with real-world voice • Adds artifacts
RNNoise	• Great for static hums and wind • Very efficient for real-time use	• Not suitable for music or speech overlaps • Limited offline use
DeepFilterNet	• High-quality filtering • Real-time capable	• Complex setup (ONNX/C++) • Overkill for simple pipelines

Voicefixer	• Can restore low-quality recordings • Deep learning powered	• Requires tuning for degraded inputs • Not ideal for general transcription use

4.3 Smart Chunking

Chunking is critical when working with long audio files that may exceed the token or memory limits of transcription models like Whisper. Rather than splitting arbitrarily by time or token count, smart chunking uses natural pauses in speech to segment the audio — resulting in cleaner, more accurate transcription. Our implementation leverages silence detection via pydub and allows customization through the following parameters:

- min_silence_len: The minimum duration of silence (e.g., 1000ms) required to trigger a new chunk
- silence_thresh: The volume threshold (in dBFS) below which audio is considered silent
- chunk_padding: Optional buffer added before and after each chunk to preserve speech context

This approach respects sentence and phrase boundaries, reducing the risk of cutting through spoken words or thoughts. It strikes a balance between automation and linguistic sensitivity: fully unsupervised, yet more speech-aware than naïve fixed-length slicing. By preserving the natural rhythm of conversation, smart chunking improves transcription quality, enhances summarization coherence, and ensures that downstream models operate on well-formed, context-rich segments — especially crucial for long-form audio like podcasts, meetings, or interviews.

Chunking Method	Description	Pros	Cons
Smart Chunking (Silence-Based)	Detects natural pauses in audio to define chunk boundaries.	Respects sentence structure; improves contextual accuracy.	Silence detection may miss in noisy or overlapping speech.
Fixed Time Chunking	Splits audio into equal-length windows (e.g., every 30 seconds).	Easy to implement; predictable size; no preprocessing needed.	Often cuts words or phrases mid-sentence; poor for readability.
Token-Based Chunking	Splits transcript text by token count for LLM summarization.	Helps fit into model limits like 4096 tokens.	Doesn't solve audio splitting; works after transcription.
Manual Segmentation	User-defined markers or	Precise control; ideal for curated datasets.	Not scalable; requires manual effort.

	timestamps break audio		
VAD-Based Chunking	Uses Voice Activity Detection (like WebRTC or py-webrtcvad).	Effective even in noisy audio; handles speech activity better than silence thresholding.	More complex to tune; might segment too aggressively.

4.4 Whisper Transcription

Refer to Section 3

4.5 Summarization

To generate high-level, human-readable summaries from transcribed audio, our system supports a fully customizable summarization pipeline. Customers can choose any instruction-tuned large language model (LLM) from Hugging Face or supply their own locally fine-tuned model by passing the --summarized-model argument. This flexibility is one of our key innovations, allowing enterprises to align the summarization tone, language, and style with their specific domain.

By default, we use mistralai/Mistral-7B-Instruct-v0.1, a lightweight but powerful open-weight model designed for structured summarization. It provides excellent quality while remaining efficient on commonly available GPUs.

Summarization is optional and triggered only when the --summary flag is passed. When enabled, summaries are appended directly to both the .txt and .json output files alongside the raw transcript, enabling downstream search, QA, or summarization workflows.

4.6 Output

To make transcription results easy to consume and integrate, our system not only produces standardized output files, but also exposes a RESTful API interface. This combination of structured outputs and remote accessibility ensures the pipeline can serve a wide range of end users — from developers to analysts to customer-facing tools.

4.6.1 Output Formatting Strategy

For each processed audio or video input, the system generates:

- `.txt` file:Contains the full transcript in human-readable format. If --summary is enabled the summary will also be append to the file.
- `.json` file:A structured version of the transcript, including metadata such as chunk boundaries, model information, and (optionally) the generated summary.
- `.log` file:Captures detailed runtime information including processing steps, warnings, and performance metrics for debugging or audit purposes.

These outputs are organized and saved per input file, making them easy to archive or pass to downstream pipelines (e.g., summarization, search indexing, analytics).

4.6.2. API Deployment: Making Transcription Accessible

To extend usability and integration flexibility, the entire Whisper pipeline is also deployed as a REST API using FastAPI and Uvicorn. This allows users or external systems to:

- Upload audio/video files remotely via HTTP
- Trigger transcription and summarization automatically
- Retrieve structured results asynchronously

This API layer transforms the Whisper pipeline from a developer tool into a scalable transcription microservice, ready to be embedded in real-world enterprise or SaaS environments.

5. CLI Arguments and Usage

This section outlines the flexibility and innovation integrated into the Whisper batch pipeline via command-line arguments. These arguments allow customers to fully customize the behavior of the pipeline according to their needs, promoting modularity and user-centric design.

5.1 --input

Required. Specifies the path to an input audio file or a directory of audio files. This ensures that the pipeline can operate in both single-file and batch modes, supporting scalable workflows.

5.2 --model

Optional. Allows the user to override the default model selection logic (which is based on detected GPU). This gives power users control over which Whisper model to use (e.g., small, medium, large, turbo).

5.3 --output-dir

Optional. Lets users specify the directory where all output files (transcripts, summaries, JSON logs) will be stored. Provides organization and custom directory structure support.

5.4 --summarized-model

Optional. Accepts either a HuggingFace model ID or a local path to a custom summarization model. This is a key innovation allowing users to bring their own summarizers while maintaining full compatibility.

5.5 --ground-truth

Optional. Path to a file containing the ground truth transcript. If provided, the pipeline calculates the Word Error Rate (WER) of the generated transcript. Useful for benchmarking and validation purposes.

5.6 --denoise

Optional flag. When specified, the pipeline applies a two-step denoising process (Demucs + noisereduce) to enhance transcription accuracy in noisy environments like meetings or crowded spaces.

5.7 --prop-decrease

Optional. Float between 0.0 and 1.0 that controls the aggressiveness of noise suppression during denoising. Default is 0.7. This offers fine-tuned control over denoising behavior based on user preference or data quality.

5.8 --summary

Optional flag. When set, the pipeline generates a structured summary of the transcript using the chosen summarization model. This helps create concise insights from long speech data such as interviews, meetings, or podcasts.

These arguments represent core innovations that prioritize user control, adaptability, and workflow flexibility. By enabling users to tailor the behavior of the pipeline—ranging from model selection and file handling to denoising, summarization, and benchmarking—we ensure the system meets a broad range of real-world use cases.

6. Edge Case Handling, Flexibility, and Innovations in the Whisper Pipeline

This pipeline was designed from the ground up with fault tolerance, modularity, and end-user customization in mind. By handling edge cases, enabling fallback logic, and exposing key toggles via CLI, the system adapts to diverse transcription needs — from clean studio recordings to noisy real-world media — while providing high-quality, structured output. Below is a summary of key innovations and edge-case protections that make this system both flexible for users and robust for enterprise use:

6.1 Robust Error Handling & Edge Case Coverage

Mechanism	Description
Input Validation	Automatically verifies file format, sample rate, size, and media compatibility.
Video & Audio Format Support	Supports .mp3, .wav, .flac, .m4a, .webm, .mp4, .mov, .mkv, etc. — all normalized via ffmpeg to 16kHz mono WAV.
Duration Enforcement	Files exceeding max length (e.g., 60 mins) are skipped or chunked, with configurable limits.
Try/Except Blocks	Every critical operation (e.g., model inference, ffmpeg, chunking, denoising) is wrapped in exception handling to prevent full-pipeline crashes.
Skip Unusable Files	Empty, silent, or corrupt inputs are detected and skipped gracefully.
Batch & Single-File Support	Accepts either a single input file or a full directory for batch transcription.
Failure Isolation	One bad file doesn't stop the run. Errors are logged; processing continues for remaining files.
Logging System	Fine-grained logging at INFO/WARNING/ERROR levels tracks file-level operations, failures, durations, and hardware usage for easy debugging.
Runtime Profiling (optional)	Can log chunk-level and model-level performance for future optimization.

6.2 Core Innovations & Custom Enhancements

Feature	Description
Smart Chunking	Segments audio based on silence detection rather than fixed duration. Preserves sentence boundaries and prevents mid-thought cuts, improving transcription and summarization accuracy.
GPU-Aware Load Balancing	Automatically adjusts model size, chunking thresholds, and summarization behavior based on available hardware (e.g., A10, A100, H100).
Combined Denoising Pipeline	Applies two-stage noise reduction: 1) Demucs for structured noise separation (e.g., music); 2) noisereduce or DeepFilterNet for static/background noise.
Combined Export	Both .txt and .json outputs include transcript, speaker info, summaries, chunk metadata, and model info.
Language Auto-detection	Automatically detects language using Whisper's multilingual support, allowing cross-language audio processing.
Ground Truth Evaluation (Optional)	Computes Word Error Rate (WER) if --ground-truth is provided — useful for benchmarking.
Custom Summarization Trigger	Summary generation is optional and only activated via --summary, allowing performance optimization when not needed.
Model-Agnostic Design	Users can specify any Whisper model or LLM (e.g., LLaMA, Mistral, Mixtral) from Hugging Face or local storage.

REST API Deployment	The entire pipeline is deployable via FastAPI, allowing browser uploads, HTTP automation, and integration with external tools.
Prop-Decrease Control:	Users can customize denoising aggressiveness via --prop-decrease, tuning output quality vs. preservation.
Language Agnosticism:	While tuned for English summarization, the pipeline can process >90 Whisper-supported languages.
Flexible Output Paths	• All outputs (logs, JSON, TXT) are neatly stored under a configurable --output-dir with timestamped naming.

7. Infra and Tech Stack

This section outlines the hardware, libraries, and orchestration tools that power the transcription, summarization, and preprocessing pipeline. The goal is to ensure performance, modularity, and scalability across different deployment contexts (local, cloud, API).

Function	Tool / Library
Hardware	NVIDIA H100, A100, and A10 GPUs
Audio/Video Conversion	ffmpeg (CLI-based WAV normalization with fallback to `pydub`)
Transcription Engine	openai-whisper (Python) with optional multilingual support
Smart Chunking	pydub.silence for silence-based splitting + contextual padding
Denoising (Hybrid)	demucs for vocal separation + noisereduce / deepfilternet for static noise suppression
Summarization LLM	transformers + Hugging Face (overrideable via `--summarized-model`
Accuracy Evaluation	jiwer (for Word Error Rate vs. ground truth)
GPU Detection & Adaptation	torch.cuda.get_device_properties() for dynamic chunk/model scaling
Batch Orchestration	Python CLI runner (Docker-ready) + Oracle Kubernetes Engine (OKE) support
API Layer	FastAPI (REST API to support remote use cases and integrations)
Logging & Monitoring	logging module with structured INFO/WARN/ERROR messages, plus timestamped file-level logs

8. References

- OpenAI Whisper GitHub: https://github.com/openai/whisper
- Whisper model card: https://huggingface.co/openai/whisper
- OpenAI Whisper Paper: ["Robust Speech Recognition via Large-Scale Weak Supervision"](https://cdn.openai.com/papers/whisper.pdf)Whisper Benchmark: Hugging Face: https://huggingface.co/spaces/openai/whisper
- PyDub silence splitting: https://github.com/jiaaro/pydub/blob/master/API.markdown#silence
- WebRTC Voice Activity Detection (VAD): https://github.com/wiseman/py-webrtcvad
- Best practices for ASR chunking: Mozilla DeepSpeech docs, Whisper community discussionsPyannote-Audio (Speaker Diarization): https://github.com/pyannote/pyannote-audio
- Mistral AI models: https://huggingface.co/mistralai
- Open LLM leaderboard: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard
- FLAN-T5: https://huggingface.co/docs/transformers/model_doc/flan-t5
- Mixtral (MoE): https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1
- OpenAI GPT models: https://platform.openai.com/docs/models/gpt-4
- FastAPI documentation: https://fastapi.tiangolo.com/
- REST API design practices: https://restfulapi.net/
- JSON API response standards: https://jsonapi.org/
- FFmpeg docs: https://ffmpeg.org/ffmpeg.html
- yt-dlp usage: https://github.com/yt-dlp/yt-dlp
- Whisper supported formats: https://github.com/openai/whisper#requirements
- AudioSegment (PyDub): https://pydub.com/
- Demucs: https://github.com/facebookresearch/demucs

- noisereduce: https://github.com/timsainb/noisereduce
- RNNoise: [https://people.xiph.org/\~jm/demo/rnnoise/](https://people.xiph.org/~jm/demo/rnnoise/)
- Spleeter: https://github.com/deezer/spleeter
- DeepFilterNet: https://github.com/Rikorose/DeepFilterNet
- Standard argparse usage: https://docs.python.org/3/library/argparse.html
- Custom CLI flag patterns in open-source Whisper wrappers (e.g. whisperx, `faster-whisper`)