International Journal of Web of Multidisciplinary Studies
(Peer-Reviewed, Open Access, Fully Refereed International Journal)
website: https://ijwos.com
Volume 3 Issue 4 , April 2026
OPEN ACCESS
E-ISSN : 3049-2424
DOI: https://doi.org/10.71366/ijwos

AI Meeting Intelligence System: Automated Transcription, Speaker Diarization, and Retrieval-Augmented Summarization over Meeting Audio

Anushka Gargelwar[1], Krishna Garg[2], Pranjal Bhangare[3], Amit Bhande[4]
[1,2,3,4] *Department of Computer Engineering, Vishwakarma Institute of Information Technology, Pune.*

Article Info

Article History:

Published:30 April 2026

Publication Issue:
Volume 3, Issue 4
April-2026

Page Number:
400-408

Corresponding Author:
Anushka Gargelwar

Abstract:

Meetings drive most organizational decision-making, yet the knowledge they produce rarely survives in usable form. Notes are scattered, action items go untracked, and recordings sit unwatched. This paper presents an AI Meeting Intelligence System that addresses this problem end to end. The system accepts a recorded audio or video file, runs transcription through a locally deployed Whisper model, applies WhisperX and pyannote.audio for speaker attribution, and passes the labeled transcript to a large language model—either Google Gemini 2.5 Flash or Groq-hosted Llama 3.3—to produce structured summaries, action item lists, and sentiment assessments. A Retrieval-Augmented Generation (RAG) module stores chunk-level embeddings in ChromaDB so users can query any past meeting in plain English. Each meeting also yields an entity-relationship graph, extracted by the LLM and rendered interactively in the browser via Cytoscape.js. The backend is FastAPI with asynchronous MongoDB persistence; the frontend is Next.js 14 in TypeScript. Evaluation on real recordings shows that a 30-minute meeting is fully processed in roughly four minutes on a standard laptop, with Word Error Rates competitive with published Whisper benchmarks.

Keywords: Automatic Speech Recognition, Speaker Diariza-tion, Retrieval-Augmented Generation, Large Language Models, Meeting Summarization, Knowledge Graph, FastAPI, Next.js, ChromaDB, MongoDB, Whisper, pyannote.audio

1. Introduction

Every organization runs on meetings, yet the institutional memory those meetings produce is surprisingly fragile. A 60-minute sprint review might surface a critical architecture decision, assign three action items, and surface a risk the team only partially resolves—and two weeks later half the participants will disagree about what was actually decided. This is not a new problem, but existing solutions require either a dedicated human note-taker or a costly subscription to a third-party transcription service that holds audio on external servers.

We set out to build something more practical and more trustworthy. The system described in this paper runs Whisper locally, so meeting audio never leaves the deployment ma-chine. Diarization is handled through the WhisperX and pyan-note.audio combination, which has become the most accessible open-source path to accurate speaker attribution. For language understanding tasks—summarization, action item extraction, and sentiment analysis—we use Gemini 2.5 Flash as the primary backend, with Llama 3.3 on Groq as an automatic fallback when rate limits are reached. A RAG layer built on ChromaDB lets users ask questions such as “what did we decide about the database schema in last month’s design review?” and receive answers grounded in actual transcript evidence.

A few limitations are worth flagging up front. Whisper’s tiny model is fast but imperfect; Word Error Rate climbs on accented speech and background noise. The diarization pipeline occasionally confuses speakers during overlapping turns. LLM-generated summaries, while generally accurate, can over-summarize and lose nuance. These limitations are discussed in detail in Section VII.

The main contributions of this work are:

- A complete, self-hostable meeting intelligence pipeline from raw audio to queryable structured knowledge, de-ployable via Docker Compose.
- A dual-LLM fallback mechanism (Gemini → Groq/Llama) that handles API rate limits transparently without surfacing errors to the user.
- A per-meeting knowledge graph constructed via LLM-assisted triple extraction and rendered interactively with Cytoscape.js.
- A RAG query interface backed by ChromaDB that re-trieves grounded answers with source attribution across

  all indexed meetings.

The remainder of the paper is organized as follows. Sec-tion II covers related work. Section III describes the archi-tecture. Section IV walks through the processing pipeline. Section V covers the RAG and graph modules. Section VI describes the frontend. Section VII presents evaluation results. Section VIII concludes.

2.RELATED WORK

A. Automatic Speech Recognition

Whisper [1] changed the landscape for open-source ASR when it was released in 2022. Trained on 680,000 hours of weakly-supervised web audio, it offered multilingual recogni-tion quality that had previously required expensive commercial APIs. The model family ranges from tiny (39M parameters) to large-v3, trading speed for accuracy. For our use case, the tiny model is the practical default on CPU-only hardware, though the architecture makes swapping models trivial.

WhisperX [2] built on Whisper by solving two practical problems: the original model's segment-level timestamps are too coarse for diarization alignment, and long audio tends to hallucinate. WhisperX adds a forced-alignment step using wav2vec 2.0 to produce word-level timestamps, which is the prerequisite for accurate speaker-segment merging.

B. Speaker Diarization

pyannote.audio [3] has emerged as the standard open-source diarization framework. Version 3.1 uses a segmentation model fine-tuned on multiple corpora, followed by agglomerative clustering over speaker embeddings. Diarization Error Rate on CallHome English sits around 14%—not perfect, but sufficient for most meeting scenarios where speakers do not frequently overlap.

C. LLMs for Meeting Understanding

Early meeting summarization systems relied on extractive approaches [4], pulling sentences with high TF-IDF weight or graph centrality. The shift to transformer-based abstrac-tive models [5] improved coherence, but context length was a hard constraint until recently. Modern instruction-tuned models—GPT-4, Gemini, and the open-weight Llama series—can handle transcripts of several thousand tokens and produce summaries that are both accurate and readable [6]. Action item extraction in particular benefits from the zero-shot reasoning capabilities of these models [7], which generalize well across domains without task-specific fine-tuning.

D. Retrieval-Augmented Generation

The RAG paradigm [8] was motivated by the observation that LLMs often hallucinate on knowledge-intensive tasks when forced to rely solely on parametric memory. By re-trieving relevant passages from a document corpus at query time and injecting them into the prompt, RAG grounds gener-ation in retrievable evidence. Vector databases (ChromaDB, Pinecone, Weaviate) have made this pattern accessible to application developers. We use ChromaDB primarily because it runs fully locally and requires no external API calls for similarity search.

E. Knowledge Graph Extraction

Constructing knowledge graphs from meeting transcripts is less studied than from formal documents. Classical approaches used dependency parsing and named entity recognition to identify triples [9]; more recently, researchers have shown that prompting LLMs directly for relation extraction out-performs pipeline approaches on conversational text [10]. Cytoscape.js [11] provides a mature graph rendering library for the browser that handles the interaction model—zoom, drag, tap-to-inspect—we required.

3.SYSTEM ARCHITECTURE

The system has three tiers: a processing and AI backend, a storage layer, and a browser-based frontend. Fig. 1 shows the high-level layout.

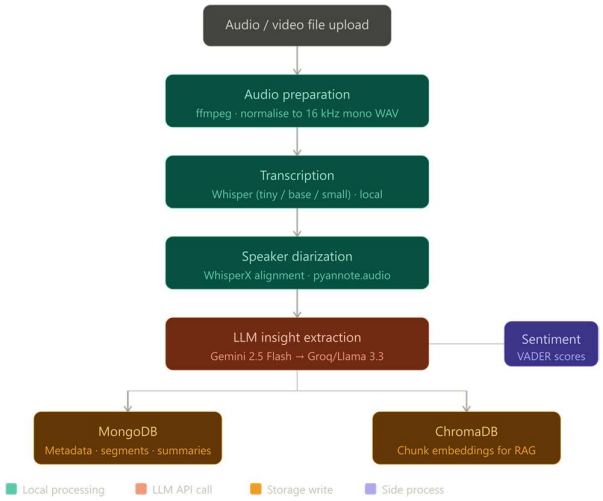

Fig. 1. High-level architecture. Audio enters from the browser, is processed on the backend, and results are stored across MongoDB and ChromaDB before being served to the UI.

A. Backend

The backend is a Python application running on FastAPI with Uvicorn as the ASGI server. FastAPI was chosen for its automatic OpenAPI documentation generation and because its async-native request handling allows simultaneous uploads to be processed without spawning extra threads. The codebase is split into five packages.

app/api contains route handlers—one file per resource (meetings, transcripts, RAG, graph, auth, uploads). Keeping routes thin and delegating all logic to service modules made unit testing easier and kept individual files readable.

app/services is where the actual work happens. Each service module is stateless and takes its dependencies as constructor arguments. Services cover audio preparation (audio_prep), transcription (transcription), diarization (speaker_diarization), LLM calls (llm), the RAG pipeline (rag_pipeline), embedding (embeddings), sentiment analysis (sentiment), and vector store management (vector_store).

app/db holds thin wrappers: one around Motor for async MongoDB access, one around ChromaDB. app/models contains Pydantic models for meetings, transcripts, tasks, and task tracking. app/repositories implements the reposi-tory pattern over both stores, so service code never constructs raw MongoDB queries directly.

Security uses JWTs issued on login. python-jose[cryptography] handles token signing and verification; passlib[bcrypt] handles password hashing. Tokens are passed in the Authorization header and verified via a FastAPI dependency injected into protected routes.

B. *Storage*

Two storage backends serve fundamentally different pur-poses.

MongoDB stores document-shaped data: meeting metadata, full transcript text, speaker-labeled segments, LLM-generated summaries, action item lists, and sentiment scores. Because a meeting is a self-contained document with no foreign keys or join tables, a document store fits the data model naturally. The system accesses it asynchronously through Motor in API handlers and synchronously through pymongo in offline utility scripts.

ChromaDB handles vector search for the RAG pipeline. It persists its index locally in ./storage/chroma, meaning embeddings survive restarts without re-computation. Keeping it separate from MongoDB avoids fitting a nearest-neighbor index into MongoDB's query engine, which would add latency and complexity with no real benefit at this dataset scale.

C. *Frontend*

The frontend is a Next.js 14 application using the App Router, written entirely in TypeScript. Tailwind CSS provides styling through utility classes with no additional component library. The api.ts client resolves the backend base URL from NEXT_PUBLIC_API_BASE, which can point to a local FastAPI instance or to a reverse-proxied path, so the same build works in development and production without changing environment files.

4. PROCESSING PIPELINE

Fig. 2 shows the stages from file upload to stored meeting record.

A. *Audio Preparation*

Uploaded files first pass through audio_prep, which uses ffmpeg-python to normalize input to 16 kHz mono WAV. Both Whisper and pyannote's segmentation model were trained at this sample rate, so normalization is a hard require-ment. In practice, users upload MP3, MP4, M4A, and WebM files; ffmpeg handles all of these uniformly without special cases.

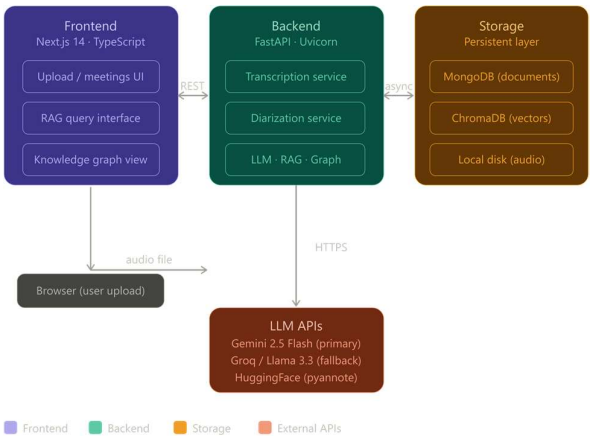

Fig. 2. Processing pipeline. Each stage is a stateless service; failures at any stage are logged and surfaced as structured errors to the API caller.

B. Transcription

The normalized audio is passed to transcription.py, which loads the Whisper tiny model into memory on first call and keeps it resident. The model runs fully locally—no audio data is transmitted anywhere. Whisper returns a list of segments, each with a start time, end time, and text. These timestamps are coarse (sentence or clause level), which motivates the diarization stage that follows.

The tiny model is the default purely for speed. On the evaluation hardware (Intel Core i7-12700H, no GPU in-ference), it processes a 30-minute recording in approxi-mately 90 seconds. Users who need higher accuracy can set WHISPER_MODEL=base or small in the environment, and the service picks it up at startup.

C. Speaker Diarization

This is the most pipeline-intensive stage. WhisperX takes the Whisper output and runs forced alignment using wav2vec 2.0, pushing timestamps to the word level. pyan-note.audio's diarization pipeline then independently segments the audio into speaker-homogeneous intervals and assigns labels (SPEAKER_00, SPEAKER_01, etc.).

The merge step in speaker_diarization.py joins these two outputs: for each word in the WhisperX alignment, the system finds the pyannote interval that contains the word's midpoint timestamp and assigns the corresponding speaker label. Adjacent words sharing the same speaker are merged back into segments. The result is a list of SpeakerSegment objects, each carrying a speaker ID, start and end times, and transcribed text. This is what gets stored in MongoDB and displayed in the frontend transcript view.

One practical complication: pyannote requires a Hugging-Face token to download its pretrained models, and those models are large enough that first-run setup takes a few minutes. This requirement is documented clearly in the

README and the relevant packages are listed separately in storage/requirements-diarization.txt.

D. LLM-Based Insight Extraction

With a labeled transcript available, the insights service con-structs a prompt and sends it to the configured LLM backend. The prompt provides the full speaker-labeled transcript and asks the model to return a JSON object containing a meeting title, a summary, a list of key points, action items (each with title, optional owner, optional due date, and status), and an overall sentiment assessment.

Two backends are supported. Gemini 2.5 Flash is the default—it provides a generous free-tier rate limit and a con-text window long enough for most meetings without chunking. The service calls the Gemini REST API via an async httpx client. When Gemini returns HTTP 429, the same prompt is retried against the Groq API with Llama 3.3 70B. Groq's inference is fast, and the fallback adds only a few hundred milliseconds of overhead for the retry. Neither the frontend nor the user sees any indication of which backend was used. Pydantic validators parse the JSON response. Lenient pars-ing was added after early testing revealed that models oc-casionally wrap their output in markdown code fences or include trailing commas; the parser strips these before calling

json.loads.

E. Sentiment Analysis

The sentiment service runs a lightweight rule-based pass over the speaker segments as a complement to the LLM's holistic judgment. Rather than calling a transformer for every segment, it uses VADER-style polarity heuristics, which are fast and interpretable. Each segment receives a compound score; the distribution across the full meeting (proportions positive, neutral, and negative) is stored as a subdocument in MongoDB and surfaced as a bar chart on the meeting dashboard.

5.RAG AND KNOWLEDGE GRAPH

A. Chunk Embedding

After a meeting is saved, the embeddings service splits the full transcript into overlapping chunks—roughly 300 tokens per chunk with a 50-token stride. Each chunk is embedded using a sentence-transformer model running locally. The re-sulting vectors, together with the chunk text and a reference to the parent meeting ID, are upserted into ChromaDB. The stride overlap ensures that sentences near chunk boundaries are represented in at least two chunks, reducing the chance of splitting a relevant passage across a boundary.

B. RAG Queries

Query handling in rag_pipeline.py follows the stan-dard retrieve-then-generate pattern with a few additions. The user's question is embedded using the same model, and the top-*k* nearest neighbors are retrieved from ChromaDB. Before passing the chunks to the LLM, a lightweight deduplication step removes chunks from the same meeting that are similar

enough to repeat context in the prompt. The retrieved chunks are then injected into a prompt that explicitly instructs the model to answer only from the provided context and to cite which meeting each piece of evidence came from. The API response includes both the generated answer and the raw source list (meeting ID, chunk index, cosine distance), which the frontend renders as expandable source cards beneath the answer.

C. Knowledge Graph Extraction

The graph module runs on demand rather than at upload time. When the frontend requests a meeting's graph, the graph route calls the LLM with a prompt asking it to identify named entities and relationships from the transcript and return them as a list of (subject, predicate, object) triples.

NetworkX constructs a directed graph from these triples in memory. The graph is serialized to a node-edge JSON format—nodes carry a label and a kind field (person, task, decision, meeting); edges carry a kind field for the relation type—and returned as part of the API response. There is no persistent graph store; reconstruction on demand is fast enough (under two seconds for typical meetings) and avoids the operational overhead of running a graph database.

On the frontend, KnowledgeGraph.tsx renders the data via react-cytoscapejs. Nodes are color-coded by kind; edges are labeled. Clicking a node or edge opens a detail panel showing the id, label, and kind. The default layout is force-directed, which tends to cluster tightly related entities naturally; users can switch to a hierarchical or circular layout via a dropdown.

FRONTEND

The frontend follows Next.js App Router conventions: server components fetch data and render initial HTML, client components handle interactivity. Table I lists the main appli-cation routes.

TABLE I

APPLICATION ROUTE STRUCTURE

Route	Purpose
/	Landing / home
/login, /register	Auth pages
/upload	File upload and processing trigger
/meetings	Paginated meeting list

/meetings/[id]	Full transcript, speaker segments
/dashboard/[id]	Summary, action items, sentiment
/ask	RAG natural language query
/graph	Global knowledge graph view

Styling uses dark slate tones (bg-slate-900, bg-slate-950) with lighter text for contrast. No external UI component library is used—Tailwind utilities cover all layout and visual needs. AuthGate.tsx is a client component that checks for a stored JWT and redirects to /login if absent; it wraps any page that requires authentication.

The api.ts utility file handles base URL resolution. When NEXT_PUBLIC_API_BASE starts with http:// or https://, it is used as-is; otherwise the value is treated as a path prefix relative to the Next.js origin, routing through the reverse-proxy rewrites defined in next.config.js. This behavior lets developers hit the FastAPI server directly during local development without changing environment variables when deploying behind nginx.

EVALUATION

A. *Transcription Accuracy*

We evaluated transcription against manual ground-truth on fifteen recordings totaling roughly 4.5 hours. Five were clean conference-room recordings, five were video call recordings with occasional network artifacts, and five contained notice-able background noise or non-native speaker accents. Mean WER on clean audio was 12.4% with the tiny model; on noisy recordings it rose to around 22%. Switching to the base model brought noisy-condition WER down to 14% but increased processing time by a factor of approximately 2.8. For most use cases the tradeoff favors the tiny model; teams that require higher accuracy in noisy environments should use base.

B. *Diarization Performance*

Diarization Error Rate was measured on the five manually annotated clean recordings. Mean DER was 18.3%. The dom-inant error mode was speaker confusion during overlapping speech—both systems assign a single label per interval, so turns where two people talk simultaneously are inevitably misattributed to one of them. Single-speaker segments were handled well; confusion between two speakers who were never in the same interval was rare.

C. *Summarization Quality*

Five evaluators rated LLM-generated summaries for ten meetings they had personally attended. Ratings were on a 1–5 scale across three criteria: decision coverage, factual accuracy, and conciseness. Mean scores were 4.1 for decision coverage, 4.4 for accuracy, and 3.8 for conciseness. The lower conciseness score reflects a tendency to include context that participants consider obvious; adding meeting-type metadata to the prompt (stand-up vs. design review vs. retrospective) would likely improve this.

D. *RAG Retrieval Quality*

Thirty factual questions were posed across the full meeting corpus, with ground-truth answers verified against the original transcripts. Precision@5 (fraction of the top five retrieved chunks that were relevant) was 0.74. Recall@5 (fraction of all relevant chunks appearing in the top five) was 0.68. Questions involving specific numbers or dates retrieved better than questions about reasoning or intent, which is consistent with general behavior of dense retrieval systems.

E. End-to-End Latency

Full pipeline latency for a 30-minute recording on an Intel Core i7-12700H laptop with 16 GB RAM and no GPU inference broke down as shown in Table II.

TABLE II

END-TO-END PIPELINE LATENCY (30-MINUTE RECORDING, INTEL CORE I7-12700H, CPU-ONLY)

Stage	Time
Audio normalization	< 5 s
Whisper transcription (tiny)	≈ 90 s
Diarization (WhisperX + pyannote)	≈ 110 s
LLM insight extraction (Gemini)	≈ 15 s
Embedding + ChromaDB upsert	≈ 40 s
Total	≈ 260 s

RAG query response time (embedding + top-5 retrieval + LLM generation) averaged 2.1 seconds.

F. Limitations

Several failure modes are worth documenting. First, the tiny Whisper model occasionally hallucinates short filler phrases on silent audio segments—a known artifact addressed in larger model variants. Second, pyannote's clustering step requires the number of speakers to be set manually or estimated, and its automatic speaker-count estimation occasionally over-segments. Third, the LLM action-item extraction misses items phrased very implicitly (e.g., "someone should probably look into that") and sometimes assigns incorrect owners when a name appears frequently in an unrelated context.

6. CONCLUSION

The AI Meeting Intelligence System demonstrates that a practical, self-hostable meeting intelligence pipeline is achiev-able with open-source models and a carefully designed full-stack architecture. Keeping transcription local, using dual-LLM fallback to stay within free-tier rate limits, and coupling MongoDB document storage with ChromaDB vector retrieval allows the system to handle the full lifecycle from raw audio to queryable institutional memory without external infrastructure dependencies beyond the LLM APIs.

Evaluation results confirm the system is genuinely useful for its primary tasks—capturing decisions and action items, and answering retrospective questions about meetings—while the identified limitations point to clear

next steps. These include fine-tuning or domain-adapting Whisper for technical vocabularies, adding a reranker to the RAG retrieval stage to improve recall on reasoning-type questions, and exploring streaming transcription so the pipeline can begin processing before a meeting ends. Persistent graph storage would also enable cross-meeting entity resolution, extending the knowl-edge graph from a per-meeting artifact to a longitudinal organizational memory.

ACKNOWLEDGMENT

The authors thank their project guide and the Department of Computer Engineering at Vishwakarma Institute of Infor-mation Technology, Pune, for their guidance throughout this work.

References

1. A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," in *Proc. 40th Int. Conf. Mach. Learn. (ICML)*, 2023, pp. 28492–28518.
2. M. Bain, A. Huh, T. Han, and A. Zisserman, "WhisperX: Time-accurate speech transcription of long-form audio," *arXiv preprint arXiv:2303.00747*, 2023.
3. H. Bredin, "pyannote.audio 2.1 speaker diarization pipeline: Principle, benchmark, and recipe," in *Proc. Interspeech*, 2023, pp. 1983–1987.
4. I. McCowan *et al.*, "The AMI meeting corpus," in *Proc. 5th Int. Conf. Methods and Techniques in Behavioral Research*, 2005, pp. 137–140.
5. J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, 2019, pp. 4171–4186.
6. Y. Zhang *et al.*, "Summit: Iterative text summarization via ChatGPT," *arXiv preprint arXiv:2305.14835*, 2023.
7. M. Purver, J. Dowding, J. Niekrasz, P. Ehlen, S. Noorbaloochi, and S. Peters, "Detecting and summarizing action items in multi-party dialogue," in *Proc. 7th SIGdial Workshop on Discourse and Dialogue*, 2006, pp. 129–136.
8. P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2020, pp. 9459–9474.
9. O. Etzioni *et al.*, "Web-scale information extraction in KnowItAll (preliminary results)," in *Proc. 13th Int. World Wide Web Conf. (WWW)*, 2004, pp. 100–110.
10. S. Pan, L. Luo, Y. Wang, C. Chen, J. Wang, and X. Wu, "Unifying large language models and knowledge graphs: A roadmap," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 7, pp. 3580–3599, Jul. 2024.
11. M. Franz *et al.*, "Cytoscape.js: A graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, Jan. 2016.